OpShin - Language analysis Report

Table of Contents

Introduction to OpShin

OpShin is a programming language for developing smart contracts on the Cardano blockchain. It's syntax is 100% valid Python code and it aims to lower the barrier to entry for Cardano smart contract development. OpShin presents itself as a restricted version of Python, written specifically for smart contract development on the Cardano blockchain. While it encourages developers to write code as they would in standard Python programs, it's important to note that not all Python features are available in OpShin.

OpShin ensures that contracts evaluate on-chain exactly as their Python counterpart. OpShin's compiler ensures that if a program successfully compiles, it meets two criterias. First, the source code is guaranteed to be a valid Python program. Second, It ensures the output running it with python is the same as running it on-chain.

Limitations

The OpShin language is a subset of python, having the following limitations:

- User-defined symbols can only be imported using `from <pkg> import *`. `import <pkg>` isn't supported.
- Mutual recursion isn't supported
- Classes can't inherit
- Tuples can't contain heterogenous types
- Containers can't contain function values
- Compiler errors are throw immediately when encountered instead of being collected
- ...

Deviations from python

The limitations of OpShin don't invalidate the claim that it is a subset of python. OpShin however deviates slightly from python, making it not strictly a subset of python:

- ...

Type System

One of the limitations of using Python as-is for smart contract development is that it is dynamically typed. The type system of OpShin is much stricter than the type system of Python. OpShin addresses this by introducing a strict type system on top of Python. What OpShin does is have an independent component called the 'aggressive static type inferencer', which can infer all types of the Python AST nodes for a well chosen subset of Python.

The class `AggressiveTypeInferencer` in the file `type_inference.py` employs a set of conditions to infer types throughout the Python code. These rules form the backbone of the type inference system, enabling type checking for each of the variables involved. As per ATI, types are resolved by flow-insensitivity and type consistency. Flow-insensitivity ignores control flow, allowing variables to retain a union of types across different points in a scope. Type consistency ensures that variables maintain the same type throughout their scope, even when conflicting information appears. When inconsistencies arise, ATI resolves them by considering the broader context and applying a consistent type across the scope.

So in simple terms every variable in OpShin has a type. There are no opaque types in OpShin, everything can be deconstructed.

Rule Category	Description
Annotated Types	Explicit type annotations are respected and used as the definitive type.
Class Type Inference	Classes must have a CONSTR_ID attribute defined as an integer to uniquely identify them.
Function Type Inference	Functions are typed based on their input parameters and return annotations.
Literal Type Inference	Literal values (integers, strings, booleans) are assigned their corresponding types.
Operator Type Inference	Binary operations are typed based on their operands.
Comparison Type Inference	Comparison operations always result in a boolean type.
List Type Inference	Lists are typed based on their elements.
Dictionary Type Inference	Dictionaries are typed based on their key and value types.

Rule Category	Description
Attribute Access Type Inference	Attribute access is typed based on the object's type and the attribute being accessed.
Function Call Type Inference	Function calls are typed based on the function's return type.
Control Flow Type Inference	The type of a variable after an if-else block is a union of types from both branches.
Loop Type Inference	Variables in loops are typed based on the inferred iterable element type.
Conflicting Types	TypeInferenceError is raised if there are conflicting types

Currently, OpShin supports only Lists and Dicts. It does not support tuples and generic types, which we see as a limitation, as these can be really valuable when writing smart contracts. This limitation of not supporting tuples and generic types might require workarounds to achieve the desired functionality.

Compilation and Execution

As part of the compilation pipeline, there are a bunch of additional rewrites, all the types are resolved through aggressive types inference and some optimizations are performed, and finally the compiler gets the simplified type annotated AST, and then translates it to `pluto`, which is the intermediate language and then compiled again to `UPLC`.

OpShin provides a toolkit to evaluate the script in Python, compile the script to `UPLC`, and compile the script to `pluto`, an intermediate language for debugging purposes and to build artifacts.

It offers a straightforward API to compile, load, apply parameters and evaluate smart contracts locally. The build process creates all required files for integration with off-chain libraries like pycardano and LucidEvolution. Key features include the ability to build validators from Python files, apply parameters during or after compilation, store and load compilation artifacts, and access important contract information such as addresses and blueprints.

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 as the python internals.

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 tranformed 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:

   ```
   <tmp-var> = <tuple-expr>
   a = <tmp-var>[0]
   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 changes 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 amed decorated, 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

Quantitative Metrics

Metrics using Gastronomy

We analysed the `UPLC` code generated by OpShin for a sample validator which adds number 1 to the input that is passed, using `Gastronomy` as the `UPLC` debugger.We also examined the intermediate language, Pluto, during the process.

```
def validator(n : int)-> int:
    return n + 1
```

Below is the Pluto output for the validator function:

```
(\1val_param0 ->
  (let
    n_1 = (# (Error ((! Trace) 'NameError: n' ())));
    validator_0 = (# (Error ((! Trace) 'NameError: validator' ())))
    in
      (let
        validator_0 =
        (# (\n_1 -> ((\1self 1other -> (AddInteger 1self 1other)) (! n_1) uplc[(con integer
           )
        ) in
        ( IData
          (let
            1p0 = (UnIData 1val_param0)
          in
            ((! validator_0) (# 1p0))
          )
        )
      )
  )
)
```

The two variables `n_1` and `validator_0` represents the variable `n` and validator name `validator` in the function respectively and are not removed as part of the optimizations.

Below is the UPLC output for the validator function:

```
(lam
  1val_param0
  [
    (lam
      n_1
      [
        (lam
          validator_0
          [
```

```
        (lam
          validator_0
          [
            (builtin iData)
            [
              (lam 1p0 [ (force validator_0) (delay 1p0) ])
              [ (builtin unIData) 1val_param0 ]
            ]
          ]
        )
        (delay
          (lam
            n_1
            [
              [
                (lam
                  1self
                  (lam 1other [ [ (builtin addInteger) 1self ] 1other ])
                )
                (force n_1)
              ]
              (con integer 1)
            ]
          )
        )
      ]
    )
    (delay
      [
        (lam _ (error ))
        [
          [ (force (builtin trace)) (con string "NameError: validator") ]
          (con unit ())
        ]
      ]
    )
  ]
)
(delay
  [
    (lam _ (error ))
    [
      [ (force (builtin trace)) (con string "NameError: n") ] (con unit ())
    ]
  ]
)
```

```
  ]
)
```

The two lambda functions, `n_1` and `validator_0`, correspond to the variables named `n` and `validator`, respectively. However, these variables are not being used. While OpShin supports various levels of optimization, which typically removes dead variables and constants, these particular variables are not removed due to the way OpShin is designed. This behavior ensures that these variables remain accessible, but it may also lead to larger script sizes than necessary.

Code Coverage Percentage

We conducted a code coverage analysis for the OpShin project using the `pytest-cov` tool. Code coverage is a metric that helps to understand which parts of the codebase are exercised by the test suite, allowing us to identify untested areas.

The following details shows the results of the code coverage assessment:

```
---------- coverage: platform linux, python 3.10.12-final-0 ----------
```

Name	Stmts	Miss	Branch	BrPart	Cover
opshin/__init__.py	12	2	0	0	83%
opshin/__main__.py	284	178	122	12	32%
opshin/bridge.py	32	11	22	3	52%
opshin/builder.py	226	36	92	8	83%
opshin/compiler.py	367	13	171	9	96%
opshin/compiler_config.py	19	0	4	0	100%
opshin/fun_impls.py	78	1	28	1	98%
opshin/ledger/__init__.py	0	0	0	0	100%
opshin/ledger/api_v2.py	189	0	76	0	100%
opshin/ledger/interval.py	55	0	18	1	99%
opshin/optimize/__init__.py	0	0	0	0	100%
opshin/optimize/optimize_const_folding.py	191	15	53	5	90%
opshin/optimize/optimize_remove_comments.py	9	0	2	0	100%
opshin/optimize/optimize_remove_deadvars.py	128	2	44	1	98%
opshin/optimize/optimize_remove_pass.py	7	0	0	0	100%
opshin/prelude.py	46	26	26	0	33%
opshin/rewrite/__init__.py	0	0	0	0	100%
opshin/rewrite/rewrite_augassign.py	11	0	0	0	100%
opshin/rewrite/rewrite_cast_condition.py	30	0	2	0	100%
opshin/rewrite/rewrite_comparison_chaining.py	15	0	4	0	100%
opshin/rewrite/rewrite_empty_dicts.py	17	1	4	1	90%
opshin/rewrite/rewrite_empty_lists.py	17	1	4	1	90%
opshin/rewrite/rewrite_forbidden_overwrites.py	12	0	2	0	100%
opshin/rewrite/rewrite_forbidden_return.py	9	0	0	0	100%

opshin/rewrite/rewrite_import.py	71	3	20	6	90%
opshin/rewrite/rewrite_import_dataclasses.py	27	1	8	1	94%
opshin/rewrite/rewrite_import_hashlib.py	39	1	10	0	98%
opshin/rewrite/rewrite_import_integrity_check.py	30	0	6	1	97%
opshin/rewrite/rewrite_import_plutusdata.py	25	0	2	0	100%
opshin/rewrite/rewrite_import_typing.py	37	3	24	1	87%
opshin/rewrite/rewrite_import_uplc_builtins.py	32	2	14	3	89%
opshin/rewrite/rewrite_inject_builtin_constr.py	16	1	4	1	90%
opshin/rewrite/rewrite_inject_builtins.py	17	0	4	0	100%
opshin/rewrite/rewrite_orig_name.py	30	0	8	0	100%
opshin/rewrite/rewrite_remove_type_stuff.py	20	0	4	0	100%
opshin/rewrite/rewrite_scoping.py	113	0	32	0	100%
opshin/rewrite/rewrite_subscript38.py	8	1	0	0	88%
opshin/rewrite/rewrite_tuple_assign.py	31	0	10	0	100%
opshin/std/__init__.py	0	0	0	0	100%
opshin/std/bitmap.py	41	0	8	0	100%
opshin/std/builtins.py	96	31	0	0	68%
opshin/std/fractions.py	89	0	38	0	100%
opshin/std/hashlib.py	1	1	0	0	0%
opshin/std/integrity.py	3	3	0	0	0%
opshin/std/math.py	24	0	8	0	100%
opshin/type_impls.py	754	83	495	75	84%
opshin/type_inference.py	811	35	373	20	94%
opshin/typed_ast.py	113	1	0	0	99%
opshin/util.py	192	19	60	2	87%
TOTAL	4374	471	1802	152	87%

Manual Review Findings

The document herein is provided as an interim update detailing the findings of our ongoing audit process on the OpShin repository. It is crucial to understand that this document does not constitute the final audit report. The contents are meant to offer a preliminary insight into our findings up to this point and are subject to change as our audit progresses.

Summary of Current Findings Across Categories

1. Security - 0
2. Performance - 2 (01, 05)
3. Maintainability - 3 (02, 03, 04)
4. Others - 0

Findings and Recommendations for Improvements

Finding 01 - Improving Error Clarity

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. 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. These enhancements would make the tool more accessible, especially for developers new to OpShin or smart contract development on Cardano.

Recommendation

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

When this command is executed in the CLI

```
`opshin eval spending examples/smart_contracts/gift.py "{\"constructor\": 0,\"fields\":[
{\"bytes\": \"1e51fcdc14be9a148bb0aaec9197eb47c83776fb\"}]}" "None" d8799fd8799f9fd8799f
```

Error Encountered:

```
`ValueError: Expected hexadecimal CBOR representation of plutus datum but could not tran
```

The error is caused by the second argument, where “None” is passed instead of a valid Plutus data object for Nothing. The error message could be improved by providing a clear example of how to pass parameters correctly in JSON format.

Finding 02 - Attaching file name to title in ‘.json’ file

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.

Finding 03 - Pretty Print generated UPLC and Pluto

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 `Pluto` 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.

Also all builtins seem to be injected regardless of use. This makes inspecting the generated output more difficult without dead var elimination turned on. Dead var elimination might have however remove parts of code that the user actually expects to be present.

Recommendation

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

Variable names should be improved (e.g. the adhoc pattern can be made more compact smaller), and only the used builtins should be injected.

Finding 04 - Improve Documentation on optimization level

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.

Finding 05 - Effect of optimization level on build output

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 will also be 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.

Finding 06 - Lack of namespaced imports

Categories: *Usability/Critical* and *Performance/Critical*

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

```
@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 show a warning indicating that the name is shadowed.

Scenario 4

The code checks for the length of imports, empty asnames, and * as a name (lines 76–84), but it does not check for duplicate imports. This allows the same module to be imported multiple times without warnings or errors.

```
from typing import Dict, List, Union
from typing import Dict, List, Union
from typing import Dict, List, Union
```

These imports can be given any number of times, leading to redundant code.

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.

Finding 07 - Compiler version inconsistency

Category: *Maintainability/Minor*

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

Recommendation

According to stackoverflow, the following change to `__init__.py` might be enough:

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

Finding 08 - Migrate some utility functions

Maintainability/Informational

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.

Finding 09 - PlutoCompiler.visit_Pass is redundant

Category: *Maintainability/Informational*

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.

Finding 10 - Rewriting chained comparisons doesn't create copies of middle expressions

Categories: *Maintainability/Major* and *Performance/Minor*

When rewriting `<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 lead to difficult to debug issues 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).

Finding 11 - Compiler step 22 doesn't do anything

Category: *Maintainability/Major*

Compiler step 22 is supposed to inject `bool()`, `bytes()`, `int()`, and `str()` builtins as RawPlutExprs, but the internal types (i.e. `.constr_type()`) of those functions is inherently polymorphic (i.e. `PolymorphicFunctionType`), which is immediately skipped. 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`.

Finding 12 - Type safe tuple unpacking

Category: *Usability/Major*

Tuple unpacking (step 7) 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 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.

Finding 13 - Non-friendly error message in AggressiveTypeInferencer.visit_comprehension

Category: *Usability/Minor*

Error message on line 1185 of `opshin/type_inference.py` claims "Type deconstruction in for loops is not supported yet". But such for-loop specific deconstructions should be ok as they were rewritten in compiler step 7.

Recommendation

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

Finding 15 - Incorrect hint when using Dict[int, int] inside Union

Category: *Usability/Minor*

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`.

Finding 16 - Incorrect hints when using opshin eval incorrectly

Category: *Usability/Minor*

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".

Why does the first hint propose using `opshin eval lib`?

Recommendation

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

Finding 17 - Non-friendly error message when using wrong import syntax

Category: *Usability/Major*

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.

Finding 18 - Implicit import of plt in compiler.py

Category: *Maintainability/Minor*

In `compiler.py`:

- `plt` is made available by import 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.

Findings 19 - Irrelevant UPLC builtins in output

```
def validator(datum: bytes, redeemer: None, context: ScriptContext) -> None:
    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 like addInteger, lessThanInteger, and lengthOfByteString, which seems irrelevant while the logic is to access the first byte of the datum(`ByteString`) and to check if its equal to 0.

Findings 20 - Determinisim of Constructor Ids

```
@dataclass
class DatumOne(PlutusData):
    CONSTR_ID = 0
    inttype: int

@dataclass
class DatumTwo(PlutusData):
    CONSTR_ID = 1
    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 in Union types should be maintained. Additionally, it could be expanded to include a warning or error if no `CONSTR_ID` is provided, to alert developers about relying on automatically generated IDs.

Findings 21 - Function `to_cbor_hex()` not working

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

Findings 22 - Relative Imports Not Supported

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.

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

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

def validator():
    pass
```

```
# 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.

Findings 23 - Assumption of spec for the Parent Module in rewrite/rewrite_import.py

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.

Findings 24 - Iterating Over `sys.modules` Safely

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.

Finding 25 - Nested Lists Not Handled Correctly

The following program throws an error

```
from typing import Dict, List, Union

def validator()-> List[List[int]]:
    empty_List : List[List[int]] = [[]]
    return empty_List
```

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 improper handling of nested structures.

Finding 26 - Error Messages Are Not Descriptive in Rewrite transformers

The error messages for assertions in most of the rewrite transformers 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:

```
from pycardano import Datum as Anything, PlutusData
from dataclasses import dataclass as dc

@dc
class MyClass(PlutusData):
    pass

def validator() :
    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.

```
from dataclasses import dataclass as dc
AssertionError: The program must contain one 'from dataclasses import dataclass'
Note that opshin errors may be overly restrictive as they aim to prevent code with unintende
```

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

Finding 27 - Custom Function declarartions are Overridden

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:

```
from dataclasses import dataclass

# Custom dataclass decorator
def dataclass(cls):
  return cls

# Refers to the custom decorator, not the one from 'dataclasses'
@dataclass
class MyClass(PlutusData):


def validator(a: int) -> None:
  return None
```

The code checks for the presence of the `@dataclass` decorator and validates dataclass is imported from the package `dataclasses` but does not verify/report if the decorator is overridden by a custom dataclass 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 exisitng import".

Finding 28 - Alias names for imports

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

1.rewrite/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 like the code below, it throws a type inference error. It does not provide a clear or user-friendly error message about the name conflict.

```
from hashlib import sha256 as hsh
```

```
    x = hsh(b"123").digest()
    hsh = b"opshin"
```

2.rewrite/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).

Finding 29 - Importing Self from typing does not work

1. When attempting to import Self from the typing module, the following error occurs:

```
from typing import Self

from opshin.prelude import *

@dataclass
class MyClass(PlutusData):
    def method1(self) -> Self:
        pass
```

```
ImportError: cannot import name 'Self' from 'typing' (/usr/lib/python3.10/typing.py)
```

2. The `visit_classDef` method in `rewrite/rewrite_import_typing.py` currently replaces self with the class name, likely for type-checking purposes. However, it only checks for Name and Union types. It does not handle cases where Self is used in types, such as List[Self] or Dict[str, Self].

Finding 30 - using List or Dict directly as function argument types throws a non user-friendly error

Category: *Usability/Minor*

Newcomers to Opshin might try the following syntax:

```
from opshin.prelude import *

def validator(_: List) -> None:
    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`:

```
from opshin.prelude import *

def validator(_: Dict) -> None:
    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.

Finding 31 - bytes.fromhex() doesn't work

Category: *Usability/Major*

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

The following snippet doesn't compile though:

```
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.

Finding 32 - non user-friendly error when using Union of single type

Category: *Usability/Minor*

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

```
from opshin.prelude import *

@dataclass()
class A(PlutusData):
    x: int

def validator(a: Union[A]) -> None:
    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.

Finding 33 - inconsistent treatment of duplicate entries in Union

Category: *Usability/Minor*

Duplicate entries in `Union`s give compiler errors, but duplicate entries in nested `Union`s don't.

Consider the following example validator:

```
from opshin.prelude import *

@dataclass()
class A(PlutusData):
    x: int

@dataclass()
class B(PlutusData):
    x: bytes

def validator(a: Union[A, A, B]) -> None:
    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:

```
from opshin.prelude import *

@dataclass()
class A(PlutusData):
```

```
    x: int

@dataclass()
class B(PlutusData):
    x: bytes

def validator(a: Union[A, Union[A, B]]) -> None:
    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 being used, which might lead to unexpected duplicate entries in `Union`s.

Optionally a compiler step can be added to detect duplication of unresolved names in a single level of a `Union`, which might point to the user having made a mistake.

Finding 34 - Unions can contain classes with same CONSTR_ID if their fields are also the same

Category: *Usability/Major*

The following example validator compiles successfully:

```
from opshin.prelude import *

@dataclass()
class A(PlutusData):
    CONSTR_ID = 1
    a: bytes

@dataclass()
class B(PlutusData):
    CONSTR_ID = 2
    a: int
    b: int

@dataclass()
class C(PlutusData):
    CONSTR_ID = 2
    a: int
    b: int

def validator(_: Union[Union[A, B], C]) -> None:
    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:

```
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_IDs 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.

Finding 35 - can't use empty literal dicts in arbitrary expressions

Category: *Usability/Minor*

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:

```
from opshin.prelude import *

def my_len_fn(d: Dict[Anything, Anything]) -> int:
    return len(d)

def validator(_: None) -> None:
    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:

```
def validator(_: None) -> None:
    d = {}
    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).

Finding 39 - calling `str()` on a Union gives a non user-friendly error

Category: *Usability/Major*

Consider the following example validator:

```
def validator(a: Union[int, bytes]) -> None:
    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]`.

Finding 40 - unable to loop over tuple

Category: *Maintainability/Major*

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

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

Instead the compiler throw 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()`.

Finding 41 - list and dict comprehensions don't check that filters evaluate to boolean types

Category: *Security/Critical*

The list comprehension type checks in `AggressiveTypeInferencer.list_comprehension()` doesn't 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 by thrown if the comprehension generator returns a non-empty list. This can lead to a dead-lock of 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:

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

Recommendation

Wrap list comprehension `ifs` with Bool casts in `rewrite_cast_condition.py`.

Finding 42 - eval_uplc doesn't handle errors in ComputationResult correctly

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 from finding 41, gives the following output:

```
Starting execution
------------------
Execution succeeded
Traceback (most recent call last):
  File "/home/user/.cache/pypoetry/virtualenvs/opshin-Gqoty4Xw-py3.9/bin/opshin", line 6, in
    sys.exit(main())
  File "/home/user/Src/Opshin/opshin/opshin/__main__.py", line 518, in main
    perform_command(args)
  File "/home/user/Src/Opshin/opshin/opshin/__main__.py", line 416, in perform_command
    ret = uplc.dumps(ret.result)
  File "/home/user/.cache/pypoetry/virtualenvs/opshin-Gqoty4Xw-py3.9/lib/python3.9/site-pack
    return u.dumps(dialect)
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.

Finding 43 - Dict with Union type key, can't be accessed with a Union type which has the same entries but in a different order

Category: *Usability/Minor*

Consider the following validator:

```
from opshin.prelude import *

def validator(d: Dict[Union[int, bytes], int]) -> int:
    key: Union[bytes, int] = 0
    return d[key]
```

Compiling this example throws the following error: `Dict 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.

Finding 44 - inconsistent type inference of literal lists and dicts

Category: *Usability/Major*

The following is valid Opshin:

```
a: Union[int, bytes] = 10
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:

```
a: Union[int, bytes] = 10
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.

Finding 45 - type assertion wrappers not applied in while statement bodies

Category: *Security/Critical*

In `AggressiveTypeInferencer.visit_While()`, type assertions performed in the while statement condition don't result in the addition of Pluto 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:

```
from opshin.prelude import *

def validator(a: Union[int, bytes]) -> None:
    while (isinstance(a, int)):
        if (a > 0):
            a -= 1
```

Recommendation

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

Finding 46 - type assertion wrappers not applied on the right-hand-side of BoolOp

Category: *Security/Critical*

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

Similar to finding 44, 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 left-hand-side of the `and` expression is true:

```
from opshin.prelude import *
```

```python
def validator(a: Union[int, bytes]) -> None:
    assert isinstance(a, int) and a == 10
```

Recommendation

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

Finding 47 - wrong return type annotation of some TypeCheckVisitor methods

Category: *Maintainability/Informational*

`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`.

Finding 48 - error-prone implementation of `scopes` and `wrapped` in AggressiveTypeInferencer

Category: *Maintainability/Major*

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.

Finding 49 - resetting of `self.wrapped` in AggressiveTypeInferencer can be refactored into a separate method and simplified

Category: *Maintainability/Informational*

`visit_IfExp()` and `visit_If()` (and once finding 45 is resolved, `visit_While()`) contain the following (duplicate) lines of python code:

```python
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

Refactor the code the reverts `self.wrapped` into a new method of `AggressiveTypeInferencer`, and replace `prevtyps.keys()` by `prevtyps`.

Finding 50 - redundant code in AggressiveTypeInferencer

Category: *Maintainability/Informational*

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`.

Finding 51 - rewrite of dunder override of `not in` AggressiveTypeInferencer is spread over multiple methods

Category: *Maintainability/Informational*

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 responsability of the callsite.

Finding 52 - omitting class method return type gives non user-friendly error

Category: *Usability/Minor*

Consider the following example validator:

```
from opshin.prelude import *

@dataclass()
class MyClass(PlutusData):
    def my_method(self):
        pass
```

```
def validator(_: None) -> None:
    c = MyClass()
    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.

Finding 53 - list item that was just appended accessed immediately after

Category: *Maintainability/Informational*

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.

Finding 54 - inconsistent treatement of tuple slicing

Category: *Maintainability/Informational*

Has `AggressiveTypeInferencer.visit_Subscript()` allows tuples to be sliced, but `PlutoComplier.visit_Subscript()` doesn't.

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)`.

Finding 55 - `eval_uplc` ignores print()

Category: *Usability/Minor*

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.

Finding 56 - RecordReader.extract() doesn't need to be static

Category: *Maintainability/Informational*

`RecordReader.extract()`, in `type_inference.py`, is static has the `@classmethod`. 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`).

Finding 57 - TypedModule Dependency Before Type Inference

Category: Maintainability/Minor

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.

Finding 58 - Inconsistent Handling of Polymorphic Functions

Category: Maintainability/Minor

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:

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

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

```
if isinstance(typ.typ, PolymorphicFunctionType):
    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.

Finding 59 - Redundant Explicit Cast to Boolean

Category: Performance/Minor

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 transformation is redundant when:

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.

Finding 60 - Inability to Assign to List Elements in Validator Functions

Category : Functionality/Minor

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.

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

Finding 61- Annotated Variable Nodes Not Handled in `rewrite/rewrite_orig_name.py`

Category: Maintainability / Minor

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 may also contain a pointer to the original name for good.

Recommendation

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

Finding 62 - `NameError` expressions are added for each loaded variable

Category: *Performance/Major*

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 in the case a variable is referenced that doesn't actually exist. But the compiler should be able to detect such situations beforehand anyway, thus this should never actually occur during runtime.

The Opshin Pluto->UPLC compilation step isn't able to eliminate these `NameError` expressions, even at optimization level 3.

Recommendation

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

Finding 63 - It isn't clear when a Python `Constant` can be `PlutusData`

Category: *Maintainability/Informational*

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.

Finding 64 - unnecessary identity function wrapping in annotated assignment when assigning data to data (i.e. `Anything` to `Anything`)

Category: *Performance/Minor*

In `PlutoCompiler.visit_AnnAssign()` in `compiler.py`, data values on the right-hand-side are implicitly converted 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 doesn't have much overhead as it results in two wrapped identity functions during the code generation, but it is still unnecessary.

Recommendation

Don't perform any implicit conversions if both the right-hand-side and the left-hand-side are data values.

Finding 65 - `UnionType` not implicitly converted

Category: *Security/Critical*

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:

```
from opshin.prelude import *

def convert(a: int) -> Union[int, bytes]:
    return a

def validator(a: Union[int, bytes]) -> Union[int, bytes]:
    if isinstance(a, int):
        # In the following the typechecking assumes the return type is `Union[int, bytes]`,
        # but on-chain it will still be `int` due to missing conversion
        b = convert(a)
        if isinstance(b, int):
            print(str(b))

    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()`.

Finding 66 - redundant passing of all possible bound external variables when calling functions

Category: *Performance/Major* (perhaps *Performance/Critical* ??)

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`:

```
def add(a: int, b: int) -> int:
   return a + b

def validator(a: int, b: int) -> int:
   return add(a, b)
```

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

```
(\
  1val_param0 1val_param1 -> (
```

```
    let
      a_1 = (# (Error ((! Trace) 'NameError: a' ())));
      a_2 = (# (Error ((! Trace) 'NameError: a' ())));
      add_0 = (# (Error ((! Trace) 'NameError: add' ())));
      b_1 = (# (Error ((! Trace) 'NameError: b' ())));
      b_2 = (# (Error ((! Trace) 'NameError: b' ())));
      validator_0 = (# (Error ((! Trace) 'NameError: validator' ())))
    in (
      let add_0 = (# (
        \a_1 b_1 -> (
          (\
            1self 1other -> (AddInteger 1self 1other)
          ) (! a_1) (! b_1)
        )
      )) in (
        let validator_0 = (# (\
          add_0 a_2 b_2 -> (
            let
              1p0 = (! a_2);
              1p1 = (! b_2)
            in (
              (! add_0) (# 1p0) (# 1p1)
            )
          )
        )) in (
          IData (
            let
              1p0 = (UnIData 1val_param0);
              1p1 = (UnIData 1val_param1)
            in (
              (! validator_0) add_0 (# 1p0) (# 1p1)
            )
          )
        )
      )
    )
  )
)
```

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

Recommendation

Opshin doesn't 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.

Finding 67 - `plt.ConstrData(plt.Integer(0), plt.EmptyDataList())` appears in several places

Category: *Maintainability/Informational*

`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.

Finding 68 - unable to use negative index subscripts

Category: *Maintainability/Minor* (or *Usability/Minor* ??)

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

Finding 69 - out-of-range tuple index throws a non user-friendly error

Category: *Usability/Minor*

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 Pluto codebase.

Finding 70 - `TypedSubscript.slice.lower` and `TypedSubscript.slice.upper` don't exclude `None`

Category: *Maintainability/Informational*

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`.

Finding 71 - key data value conversion is loop invariant

Category: *Performance/Minor*

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

Recommendation

Assign `transform_output_map(dict_typ.key_typ)(OVar("key"))` to a temporary variable and move it out of the loop.

Finding 72 - almost every user-defined variable requires `Force/Delay`

Category: *Performance/Major*

Notably in `PlutoCompiler.visit_ClassDef()` in `compiler.py`, the class constructor function is wrapped in a `Delay` term. This is unnecessary as it simple a

`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 overhead almost everywhere in the Opshin generated UPLC.

Recommendation

Don't require wrapping with `Delay`/`Force` for UPLC variables containing Lambda functions. The Opshin AST should contain enough type information to be able to detect when a user-defined variable refers to a Lambda function or not.

Finding 73 - `PlutoCompiler` `visit_ListComp()` and `visit_DictComp()` are mostly the same

Category: *Maintainability/Minor*

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()`.

Finding 75 - wrong type annotation in `Type.binop` and `Type._binop_bin_fun`

Category: *Maintainability/Minor*

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`.

Finding 76 - the `CONSTR_ID` attribute is defined for `Anything`

Category: *Usability/Minor*

The following is valid Opshin, but is conceptually strange as it isn't consistent with how attributes are exposed of regular `Unions` (they must exist on each subtype), and can lead to unexpected runtime errors:

```
from opshin.prelude import *

def validator(l: List[Anything]) -> int:
    return l[0].CONSTR_ID
```

Recommendation

Remove the `CONSTR_ID` attribute for `Anything`.

Finding 77 - `ListType.copy_only_attributes()` wrongly applies data conversion to items

Category: *Security/Critical*

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:

```
from opshin.prelude import *
from opshin.std.integrity import check_integrity

@dataclass
class A(PlutusData):
   d: List[List[int]]

def validator(d: int) -> None:
   a: A = A([[d]])
   check_integrity(a)
   pass
```

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

Recommendation

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.

Finding 78 - `RecordType.cmp()` and `UnionType.cmp()` are almost exact copies of `AnyType.cmp()`

Category: *Maintainability/Minor*

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()`.

Finding 79 - the `CONSTR_ID` attribute is defined for `Union` of primitives.

Category: *Security/Critical* (or maybe merge with finding 76, and change to *Usability/Major*)

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

```
from opshin.prelude import *

def validator(u: Union[int, bytes]) -> int:
   return u.CONSTR_ID
```

Recommendation

Don't expose the `CONSTR_ID` attribute of `Union`s which contain some non-`ConstrData` types.

Finding 80 - inconsistent naming of temporary variable in `UnionType.stringify()`

Category: *Maintainability/Informational*

In `UnionType.stringify()` in `type_impls.py`, `c` is used 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 Pluto variable containing the constructor index.

Finding 81 - `zip` is used without checking equality of lengths

Category: *Security/Critical*

In `TupleType.__ge__` in `type_impls.py`, the python builtin `zip` function is used without checking 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 runtime failures, for example:

```
def validator(a: int) -> int:
   t1 = (a, a, a)
   t2 = (a, a)

   t3 = t1 if False else t2

   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__`.

Finding 82 - the `index` method of `ListType` is incorrectly implemented

Category: *Security/Critical*

The `index` method, defined in `ListType.attribute()` in `type_impls.py`, uses the wrong builtin method to check item equality. The check is currently implemented as `EqualsInteger(x, HeadList(xs))`, which only works for lists of integers.

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

```
from opshin.prelude import *

def validator(a: Anything, b: Anything) -> int:
   l: List[Anything] = [a, b]

   return l.index(b)
```

Recommendation

Change the check to `EqualsData(transform_output_map(itemType)(x), transform_output_map(itemType)(HeadList(xs)))`.

Finding 83 - `super.binop_bin_fun()` not called

Category: *Maintainability/Minor*

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.

Finding 84 - `hex` and `oct` methods perform two loops

Category: *Performance/Minor*

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 the list of integers 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.

Finding 85 - `int` method performs two loops when parsing strings

Category: *Performance/Minor*

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.

Due to UPLC Loop overhead, merging these two loops into a single loop will give some performance benefit.

Finding 86 - typo in assertion message

Category: *Maintainability/Informational*

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"`.

Finding 87 - the `all` and `any` builtins always iterate to end of list

Category: *Performance/Minor*

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 Pluto `FoldList` function to exit the iteration prematurely when `all` or `any` encounter a `false` or `true` value respectively.

Finding 88 - the `oct` builtin is almost the same as `hex`

Category: *Maintainability/Minor*

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`.

Finding 89 - `FalseData` and `TrueData` is the wrong `CONSTR_ID`

Category: *Security/Critical*

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:

```
$(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.

Finding 90 - `POWS` always accessed in reverse order

Category: *Performance/Minor*

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

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

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

Recommendation

General Recommendations

1. Currently, there are several optimization levels and optimization-related flags. We suggest reducing this to a single optimization flag, which would make builds much easier to reproduce. If you want to keep the various options for debugging reasons, then we suggest an additional `-optimize` flag which acts as a sane default for optimization.
2. A build output that contains both unoptimized and optimized UPLC CBOR is much more useful when debugging production contracts. Though there is currently no standard format for such an output, and developers can simply generate both by running the build command twice, a single high-level command that creates a Python or TS/JS artifacts directly could improve the developer experience a lot as that is what most developers will want.
3. The conversion process to Pluto/Untyped Plutus Core (UPLC) is a complex and crucial step that could potentially contain vulnerabilities. Given its significance in the overall system, we strongly recommend prioritizing a comprehensive audit of this specific conversion process. This proactive measure would provide an additional layer of assurance.
4. Add static code analyzer (generalization of static type checker) to build process. For example: `mypy`.