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" d8799fd8799f9fd8799
```

Error Encountered:

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

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, 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.

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`

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 `PolymorphicFunctionType`, which is immediately skipped.

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 are probably other ways this can be abused to get inconsistent behavior.

Recommendation

Perform this step after type inference. Check tuple types during the 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 14 - Non-friendly error message when defining associated method on class

Category: *Usability/Minor*

Writing the following OpShin code:

```
@dataclass
class MyDatum(PlutusData):
    def my_associated_method():
        return 1
```

leads to the following error message: "list index out of range"

Recommendation

Detect that the class method has zero arguments, or that the first argument isn't `self`, and throw a better error message.

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 the `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".

So why is `opshin eval lib` even proposed in the first place?

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 used named imports in whole compiler codebase

General Recommendations

1. Currently, there are several optimizations 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.