PUBLIC

Limited Code Review

of the Vyper Compiler Front-End Type System

February 10, 2023

Produced for

by

Contents

1 Executive Summary

Dear Vyper team,

Thank you for trusting us to help Vyper with this limited code review. Our executive summary provides an overview of subjects covered in our review of the latest version of Vyper compiler according to Scope to support you in forming an opinion on their security risks.

Limited code reviews are best-effort checks and don't provide assurance comparable to a non-limited code assessment. This review was not conducted as an exhaustive search for bugs, but rather as a best effort sanity check for the pull requests of interests. The review was executed by one engineer over a period of two weeks. Given the large scope and codebase and the limited time, the findings aren't exhaustive.

The subjects covered by our review are detailed in the Review Overview section. Two pull requests, PR 2974 and PR 3182, implement a large scale refactoring, while the other PRs implement local fixes. Due to time limitations, pull request 3182 was not covered and requires further attention.

We find that the new type system implementation benefits the code in terms of readability. Some aspects of type checking are improvable, as can be seen for example in Function type_from_annotation performs no validation, HashMap are declarable outside of the storage scope or InterfaceT type comparison is incorrect for return types. Further investigation is required to cover all the changes to the type system and is likely to uncover more issues.

Focusing our attention on the other pull requests in scope, we can assert that most of the pull requests reviewed correctly implement the targeted fixes. However, some pull requests only partially implement fixes, such as Note on PR 3167: fix: codegen for function calls as argument in builtin functions, or introduce changes in semantics that need further consideration, as pointed out in Note on Pull Request 3104: refactor: optimize calldatasize check . A single pull request incorrectly implements fixes, and breaks existing compiler features (Note on PR 3211: fix: restrict STATICCALL to view).

The development of the compiler is showing substantial progress. The high number of issues uncovered make further reviews necessary, and particular attention should be given to syntactic manipulations for the validation of semantics, which are error prone as shown in Function _check_iterator_modification has false positive and false negatives , AnnAssign allows tuples assignment, Assign forbids them and HashMap variable can be left-hand of assignment if wrapped in Tuple.

The following sections will give an overview of the system and the issues uncovered. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
Medium-Severity Findings	10
Low-Severity Findings	28

2 Review Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The review was performed on the following pull requests from the `https://github.com/vyperlang/vyper` repository:

1. https://github.com/vyperlang/vyper/pull/2974
2. https://github.com/vyperlang/vyper/pull/3182
3. https://github.com/vyperlang/vyper/pull/3194
4. https://github.com/vyperlang/vyper/pull/3104
5. https://github.com/vyperlang/vyper/pull/3222
6. https://github.com/vyperlang/vyper/pull/3215
7. https://github.com/vyperlang/vyper/pull/3211
8. https://github.com/vyperlang/vyper/pull/3213
9. https://github.com/vyperlang/vyper/pull/3167

For pull request 2974 and 3182, a general review was performed based on the codebase state at commit `02339dfda0f3caabad142060d511d10bfe93c520`.

For the other pull requests, the correctness was evaluated based on a diff with the previous version.

This review was not conducted as an exhaustive search for bugs, but rather as a best effort sanity check for the pull requests of interest.

2.2 System Overview

Vyper language is a pythonic smart contract oriented language, targeting the Ethereum Virtual Machine (EVM). Vyper compiler translates the Vyper language into the EVM bytecode. The compilation process is performed in multiple phases:

1. Vyper AST is generated from Vyper source code. `vyper.ast.utils.parse_to_ast()`
    1. First, the code is pre-parsed, keywords unique to the vyper language are translated into python so that the python parse can be used. `vyper.ast.pre_parser.pre_parse()`
    2. pre-parsed code is parsed into a Python abstract syntax tree. `ast.parse()`
    3. The Python AST is annotated with token location information for the purpose of error reporting. `vyper.ast.annotation.annotate_python_ast()`
    4. The Python AST is converted in Vyper AST `vyper.ast.nodes.get_node()`
2. Literal nodes in the AST are validated. `vyper.ast.validation.validate_literal_nodes()`
3. Constants are replaced in the AST with their value. Constant expressions are evaluated. `vyper.ast.folding.fold()`

the following are looped until no replacement happens anymore:

1. `vyper.ast.folding.replace_user_defined_constants()`
2. `vyper.ast.folding.replace_literal_ops()`
3. `vyper.ast.folding.replace_subscripts()`
4. `vyper.ast.folding.replace_builtin_functions()`

4. The semantics of the program are validated. The structure and the types of the program are checked. `vyper.semantics.analysis.validate_semantics()`

    1. Module level statements (variable declarations, functions, structs, events, interface definitions ecc) are visited and validated, and added to the global scope. `vyper.semantics.analysis.module.add_module_namespace()` 2. The local execution context inside functions is type checked and validated. Types are attached to AST nodes. `vyper.semantics.analysis.local.validate_functions()`

5. Getters for public variables are added, and unused statements are removed from the AST. `vyper.ast.expansion.expand_annotated_ast()`
6. Positions in storage and code are allocated for storage and immutable variables. `vyper.semantics.analysis.data_positions.set_data_position()`
7. `vyper.codegen.global_context.GlobalContext()`
8. `vyper.codegen.module.generate_ir_for_module()`
9. `vyper.ir.optimizer()`
10. `vyper.ir.compile_ir.compile_to_assembly()`

During this review, we placed considerable attention on step 4, which includes program structure validation and type checking.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe our findings. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	1

- Note on PR 3211: Fix: Restrict STATICCALL to View

Medium-Severity Findings	10

- EnumT Does Not Implement compare_type
- Function Type_From_Annotation Performs No Validation
- Function _Check_Iterator_Modification Has False Positive and False Negatives
- HashMap Are Declarable Outside of the Storage Scope
- Interface Does Not Accept Function Names Used for Builtins
- InterfaceT Does Not Implement Type Comparison
- InterfaceT Type Comparison Is Incorrect for Return Types
- Note on Pull Request 3104: Refactor: Optimize Calldatasize Check
- Pure and View Functions Can Emit Events
- implements Statement Does Not Check Functions Mutability

Low-Severity Findings	28

- AnnAssign Allows Tuples Assignment, Assign Forbids Them
- Call to Self Check Replicated Twice in FunctionDef Analysis
- Code Duplication When Return Type of a Function Is a Tuple
- Comment Referring to Code as Dead Is Incorrect
- Comment Uses Outdated Type Classes Name
- Constant Can Be Declared With Same Name as Storage Variable
- ContractFunctionT Incorrect Namespace Argument Check
- Dead Code in _get_module_definitions
- Decorators Allowed Around Interface Functions
- Enum Members Are Not Valid as Keyword Argument Defaults
- Errors Misreport Column Offset for Vyper Preparsed Keywords
- ExprInfo for Tuple Allows Assigning to Immutables
- Function Declaration Checks if Return Type Annotation Is a Call Node
- HashMap Variable Can Be Left-Hand of Assignment if Wrapped in Tuple

- Import Level of ImportFrom Ignored
- Inaccurate Comment on TYPE_T
- Internal Functions Can Have Name Collision With Builtins
- Invalid DataLocation for Tuple ExprInfo
- Lhs of AugAssign Not Visited by _LocalExpressionVisitor
- Note on PR 3167: Fix: Codegen for Function Calls as Argument in Builtin Functions
- Pointless Assert
- Positional Only Arguments Are Allowed but Ignored in Function Definitions
- RawRevert Should Be Set as Terminal Node
- Safety Check for Bytestring Classes Not Reacheable
- Storage Location of Constants Set to Storage
- Struct Creation Without Assignment Results in Cryptic Error Message
- Tuple Node Input Does Not Work With Validate_Expected_Type
- VarInfo for self Not Constant

5.1 Note on PR 3211: Fix: Restrict STATICCALL to View

Correctness High Version 1

CS-VYPER_JANUARY_2023-001

This issue was identified in an unmerged pull request which had not been fully reviewed internally yet, and consequently had a higher likelihood of having high severity issues in it. It should not be directly compared to merged pull requests to assess the overall security of Vyper. The pull request was later abandoned.

Pull Request 3211 fails to fix the problem it is addressing. This PR would want to restrict `@pure` functions from using `staticcall`, since pure function should not be able to perform external calls, as they can't be statically checked to be pure.

Instead of forbidding `pure` functions from using `raw_call`, the PR restricts `raw_call` with `is_static_call = True` to `view` functions. Both `payable` and `nonpayable` functions should also, together with `view` functions, be allowed to perform static calls.

5.2 EnumT Does Not Implement `compare_type`

Design Medium Version 1

CS-VYPER_JANUARY_2023-002

Any enum variable can therefore the assigned to any other. This compiles but should not:

```
enum A:
    a

enum B:
    a
    b

@internal
```

```
def foo():
    a:A = B.b
```

5.3 Function Type_From_Annotation Performs No Validation

Design Medium Version 1

CS-VYPER_JANUARY_2023-003

Refactored function `type_from_annotation` introduces three vectors for type system bugs to be introduced in the compiler:

1. **The context of the annotation (data location) is not checked, which matters for HashMap and Events.**

   HashMaps should only be declared as storage variables or values of other hashmaps, and events should not be a valid type for variables, function arguments, or return types.

2. **Does not check that annotations instantiate the type correctly**

   `HashMap`, `String`, `DynArray`, and `Bytes` should always be subscripted, however this is not currently enforced.

3. **Does not check that the return value from the namespace is a valid type.**

   The last line `return namespace[node.id]`, will return any namespace element instead of only types: beside types that need subscripts, this could be VarInfos or builtins.

Stricter validation should be performed on the input and outputs of `type_from_annotation()`

5.4 Function _Check_Iterator_Modification Has False Positive and False Negatives

Correctness Medium Version 1

CS-VYPER_JANUARY_2023-004

Vyper disallows modifications to an iterator in the body of a loop through the `_check_iterator_modification` python function.

Because of how the syntactic structure is checked to perform this semantic analysis step, `_check_iterator_modification` is susceptible to both false positives and false negatives.

False negative example (this compiles but should not because `self.a.iter` is modified in the loop body):

```
struct A:
    iter:DynArray[uint256, 5]
a: A

@external
def foo():
    self.a.iter = [1,2,3]
    for i in self.a.iter:
        self.a = A({iter:[1,2,3,4]})
```

False positive example (this does not compile, but should):

```
a: DynArray[uint256, 5]
b: uint256[10]
@external
def foo():
    self.a = [1,2,3]
    for i in self.a:
        self.b[self.a[1]] = i
```

5.5 HashMap Are Declarable Outside of the Storage Scope

Correctness Medium Version 1

CS-VYPER_JANUARY_2023-005

The check that HashMaps are declared as storage variable has been suppressed after the frontend type refactor. The following is now commented out:

```
# if location != DataLocation.STORAGE or is_immutable:
#    raise StructureException("HashMap can only be declared as a storage variable", node)
```

An equivalent check has not been reinstated elsewhere. This issue originates from `type_from_annotation()` not accepting the `DataLocation` anymore.

5.6 Interface Does Not Accept Function Names Used for Builtins

Design Medium Version 1

CS-VYPER_JANUARY_2023-006

Instances of `InterfaceT` are instantiated by calling `VyperType.__init__()` and passing a list of members to be added to the type. The members are validated through `validate_identifier()`, which also checks that they do not collide with the builtin namespace. This is needlessly restricting for external interfaces. A Vyper contract will not be able to call some contracts without resorting to low level calls.

The following does not compile:

```
interface A:
    def send(): view
```

5.7 InterfaceT Does Not Implement Type Comparison

Design Medium Version 1

CS-VYPER_JANUARY_2023-007

Typechecking is not performed between interface types, a variable of any interface type can be assigned to any other interface typed variable. The reason is that `InterfaceT` does not implement a custom

`compare_type()`, and reuses the one from `VyperNode`, according to which two instances of `InterfaceT` represent the same type, regardless of their attributes.

The following should not compile, but does:

```
from vyper.interfaces import ERC20

interface A:
    def f(): view

@internal
def foo():
    a:ERC20 = A(empty(address))
```

5.8 InterfaceT Type Comparison Is Incorrect for Return Types

Design Medium Version 1

CS-VYPER_JANUARY_2023-008

Method `compare_signature` of ContractFunctionT compares two functions to check wether an interface function is implemented in the module.

To properly implement a function type, we should be able to receive *at least* whatever type could be passed as argument to the interface function, and we should return *at most* whatever could be returned by the interface function. This means that the type of arguments should be a supertype of the interface function argument type, and the type of return should be a subtype of the interface function return type.

This matters with hierarchical types, which in Vyper are `String[n]`, `Bytes[n]`, and `DynArray[type, n]`. When `m < n`, `String[m]` is a subtype of `String[n]`.

In term of a practical example, the following compiles but should not:

```
interface A:
    def f() -> String[10]: view

implements:A

@external
def f() -> String[12]:
    return '0123456789ab'
```

Somebody wanting to interact with the contract through interface `A` expects that `f()` returns at most 10 characters, however here `f()` is returning 12 characters. The order of `compare_type()` for return types should be reversed.

5.9 Note on Pull Request 3104: Refactor: Optimize Calldatasize Check

Design Medium Version 1

CS-VYPER_JANUARY_2023-009

This PR removes an unconditional check that `calldata.size >= 4` before the selector matching. It introduces an optional check that `calldata.size > 0`, which is included only if any of the selectors for the external functions is 0x00000000.

Every external function already checks that `calldata.size >= 4 + argsize`, the consequences of this PR are subtle differences in behavior when `calldata.size < 4`.

When `calldata.size < 4`, before, no selector would ever be matched, and we would end up in the fallback function. Now, when `calldata.size < 4` we could happen to match a selector ending with zeros, for example selector `0x11223300` (4 bytes) is now matched by calldata `0x112233` (3 bytes). We now execute the function, which guards against `calldata.size < 4` with an assert, which cause a REVERT. So now, instead of unconditionally going to the fallback function with `calldata.size < 4`, we either go to fallback if nothing is matched, or we revert if something is matched.

This behavior is probably not what we expect after the refactor.

An alternative possibility for optimization which could be evaluated is to remove the `calldata.size >= 4 + argsize` assert when argsize is 0, since if the `calldata.size >= 4` initial check is kept, matching any selector implies passing the assert if `argsize == 0`.

5.10 Pure and View Functions Can Emit Events

Design Medium Version 1

CS-VYPER_JANUARY_2023-010

Event emission is a state mutating operation, and causes `STATICCALL` to revert. It is therefore disallowed in `pure` and `view` functions in Solidity. In Vyper however, a `pure` or `view` function can emit events through the `log` statement, or the `raw_log()` builtin. If a `pure` or `view` external function is called, vyper will try generating a `STATICCALL` to it, but if it emits an event the function will revert.

5.11 `implements` Statement Does Not Check Functions Mutability

Design Medium Version 1

CS-VYPER_JANUARY_2023-011

the `implements` statement checks if the module's interface implements the functions in an interface, either imported or defined with an interface statement. The type of the function arguments are checked, but the mutability of functions is not considered. The mutability could be seen as a hierarchical type, where the implementing function can only have mutability equal or lower than the interface function it implements.

This compiles but should not:

```
interface A:
    def f(a:uint256): view

implements:A

@external
def f(a:uint256): #visibility is nonpayable instead of view
    pass
```

5.12 AnnAssign Allows Tuples Assignment, Assign Forbids Them

Design Low Version 1

CS-VYPER_JANUARY_2023-012

`visit_Assign` in `vyper.semantics.analysis.local` ensures that the right hand side of an assignment is not a node of type tuple, but `visit_AnnAssign` does not. Is there a rationale behind this difference of behavior?

5.13 Call to Self Check Replicated Twice in FunctionDef Analysis

Design Low Version 1

CS-VYPER_JANUARY_2023-013

Lines 113-124 of `semantics/analysis/module.py` check that a function does not call itself recursively. Line 126-144 replicate this check but with more generality, since a call to self is also a cyclic call. The check at line 113-124 is redundant.

5.14 Code Duplication When Return Type of a Function Is a Tuple

Design Low Version 1

CS-VYPER_JANUARY_2023-014

The condition at line 322 of `semantics/types/function.py` treats the case where the return annotation of a function is a tuple, getting the type by iterating over the individual tuple elements and builing the `TupleT`:

```
elif isinstance(node.returns, vy_ast.Tuple):
    tuple_types: Tuple = ()
    for n in node.returns.elements:
        tuple_types += (type_from_annotation(n),)
    return_type = TupleT(tuple_types)
```

However, calling `type_from_annotation()` directly with the `ast.Tuple` node as argument achieves the same result, using the equivalent code in `TupleT.from_annotation()`:

```
values = node.elements
types = tuple(type_from_annotation(v) for v in values)
return cls(types)
```

5.15 Comment Referring to Code as Dead Is Incorrect

Design Low Version 1

Function `types_from_BinOp` in `semantics/analysis/utils.py` contains the comment:

```
# CMC 2022-07-20 this seems like unreachable code
```

in the handling of rhs of a division/modulus operation being 0.

The code is indeed reachable. Example:

```
a:uint256

@internal
def foo() -> uint256:
    return self.a / 0
```

5.16 Comment Uses Outdated Type Classes Name

Design Low Version 1

CS-VYPER_JANUARY_2023-016

Comment at line 91 of `vyper/semantics/analysis/module.py` uses the `InterfacePrimitive` class name which has been deprecated in favor of `InterfaceT`.

5.17 Constant Can Be Declared With Same Name as Storage Variable

Design Low Version 1

CS-VYPER_JANUARY_2023-017

A constant can be declared to have the same name as a storage variable, if the constant declaration follows the storage variable declaration. However a storage variable can't be declared if a constant of the same name is already declared. This is inconsistent with what happens with immutables (can't have same name regardless of order).

This compiles:

```
a: uint256
a: constant(uint256) = 1
```

But this doesn't compile, while having the same semantics:

```
a: constant(uint256) = 1
a: uint256
```

5.18 ContractFunctionT Incorrect Namespace Argument Check

Correctness Low Version 1

CS-VYPER_JANUARY_2023-018

The following check in `ContractFunctionT.from_FunctionDef()` is redundant for contract function definitions, and wrong for interface function declarations:

```
if arg.arg in namespace:
    raise NamespaceCollision(arg.arg, arg)
```

At this point, we are still building the module namespace, so the check could pass depending on the order of module body elements.

This doesn't compile:

```
a:constant(uint256) = 1
interface A:
    def f(a:uint256): view
```

While this functionally equivalent code does compile:

```
interface A:
    def f(a:uint256): view
a:constant(uint256) = 1
```

With module function definitions, it will not compile in both cases, since the namespace is also checked in local analysis, but the error message will be different depending on order.

5.19 Dead Code in `_get_module_definitions`

Design Low Version 1

CS-VYPER_JANUARY_2023-019

In `semantics/types/user.py`, the code to validate that functions with the same name extend each other input in `_get_module_definitions()` is unused, since the same logic is already implemented in `from_FunctionDef` of ContractFunctionT. The code at lines 424-439 will never be executed, since the condition at line 424 is true, since functions with the same name are not allowed at the module level.

5.20 Decorators Allowed Around Interface Functions

Design Low Version 1

CS-VYPER_JANUARY_2023-020

In interface definitions, decorators can be used over function declarations. The decorator has however no effect on the compiler's behaviour. This compiles:

```
interface A:
    @asdfg
    def f(): view
```

5.21 Enum Members Are Not Valid as Keyword Argument Defaults

Design Low Version 1

CS-VYPER_JANUARY_2023-021

Function `check_kwargable` doesn't handle the case of `Enum` nodes. The following does not compile:

```
enum A:
    a

@external
def f(a:A = A.a):
    pass
```

5.22 Errors Misreport Column Offset for Vyper Preparsed Keywords

Correctness Low Version 1

CS-VYPER_JANUARY_2023-022

The tokenizer output is annotated with lines and columns offsets, which are then used when annotating AST nodes. Some vyper keywords such as `log`, `event`, `struct`, and `interface`, are replaced with the python keyword `class` before parsing. The `class` tokens differ in position with the vyper tokens, so the column offset is misaligned for certain errors. For example the following code (which does not compile) produces an error that misreports the position of the undeclared variable `d`.

```
event A:
    b:uint256

@external
def f():
    log A(d)
```

The following error is raised, with the ascii art arrow pointing to the wrong location:

```
UndeclaredDefinition: 'd' has not been declared.
  contract "VyperContract:7", function "f", line 7:12
       6 def f():
  ---> 7     log A(d)
  -------------------^
       8
```

5.23 ExprInfo for Tuple Allows Assigning to Immutables

Correctness Low Version 1

CS-VYPER_JANUARY_2023-023

Line 249-251 of `vyper/semantics/analysis/base.py` guard against assignment to an already assigned immutable variable. However the `is_immutable` field is left blank when creating an `ExprInfo` for a tuple (`vyper/semantics/analysis/utils.py:90-97`).

The following code generates an error in code generation, instead of typechecking:

```
c:(uint256, uint256)
d: public(immutable(uint256))
e: immutable(uint256)

@external
def __init__():
    d = 1
    e = d

@external
def f():
    d, e = self.c
```

5.24 Function Declaration Checks if Return Type Annotation Is a Call Node

Design Low Version 1

CS-VYPER_JANUARY_2023-024

Function `from_FunctionDef` of ContractFunctionT performs at line 320 of `semantics/types/function.py` the following check:

```
elif isinstance(node.returns, (vy_ast.Name, vy_ast.Call, vy_ast.Subscript)):
    return_type = type_from_annotation(node.returns)
```

However, `node.returns` has no reason to be a Call. No type annotation is a Call.

5.25 HashMap Variable Can Be Left-Hand of Assignment if Wrapped in Tuple

Correctness Low Version 1

CS-VYPER_JANUARY_2023-025

Line 249-251 of `vyper/semantics/analysis/local.py`, in `visit_Assign`, ensure that lhs of assignment cannot be a hashmap without a key. However this check is skipped if the hashmap is wrapped in a tuple.

The following code passes the check, and fails compilation during code generation, in a check considered maybe redundant.

```
a: HashMap[address, uint8]
b: HashMap[address, uint8]
c: HashMap[address, (HashMap[address, uint8], HashMap[address,uint8])]
@internal
def f():
    (self.a, self.b) = self.c[empty(address)]
```

5.26 Import Level of ImportFrom Ignored

Design Low Version 1

CS-VYPER_JANUARY_2023-026

The python `ImportFrom` ast node defines a field `level` which specifies the level of a relative import. This field is ignored in Vyper, so the following code is valid and compiles:

```
from ......................vyper.interfaces import ERC20

implements: ERC20
```

5.27 Inaccurate Comment on TYPE_T

Design Low Version 1

CS-VYPER_JANUARY_2023-027

Class `TYPE_T` is commented in `semantics/types/base.py` with

```
# A type type. Only used internally for builtins
```

The comment is inaccurate, as `TYPE_T` is also used to wrap other callable types, such as events or structs.

5.28 Internal Functions Can Have Name Collision With Builtins

Design Low Version 1

CS-VYPER_JANUARY_2023-028

If a function visibility is `@internal`, it can share its name with a builtin, if the visibility is `@external`, however, a compilation error is raised. There is no valid reason it should be so, since both kind of functions populate the `self` namespace and should behave consistently.

This compiles:

```
@internal
def block():
    pass
```

This doesn't compile:

```
@external
def block():
    pass
```

The reason is that while `skip_namespace_validation` is set to true when calling `self.add_member` in `visit_FunctionDef`, which is used for `internal` and `external` functions, it is set to false when populating the interface of the module, which includes only external functions.

5.29 Invalid DataLocation for Tuple ExprInfo

Design Low Version 1

CS-VYPER_JANUARY_2023-029

A Tuple is the only type of node whose ExprInfo is the aggregation of the individual ExprInfos of its nodes, so it is tricky to define it consistently. The ExprInfo of a Tuple containing a storage and an immutable variable has DataLocation `CODE`. This could cause problems if the `ExprInfo` was to be used in code generation.

5.30 Lhs of AugAssign Not Visited by _LocalExpressionVisitor

Correctness Low Version 1

CS-VYPER_JANUARY_2023-030

`_LocalExpressionVisitor` is a legacy class whose sole current purpose is to check that `msg.data` and `address.code` are correctly accessed in a builtin that can handle them. The left hand side of an `AugAssign` is not visited by `_LocalExpressionVisitor`, so the following passes the `_validate_address_code_attribute()` and `_validate_msg_data_attribute()` checks, and causes a compiler panic:

```
a: HashMap[Bytes[10], uint256]

@external
def foo(a:uint256):
    self.a[msg.data] += 1
```

5.31 Note on PR 3167: Fix: Codegen for Function Calls as Argument in Builtin Functions

Correctness Low Version 1

CS-VYPER_JANUARY_2023-031

PR 3167 correctly fixes an issue where arguments of builtins would be included twice in the generated code, resulting in reverts because of duplicated labels. The fix is implemented for builtins `floor`, `ceil`, `addmod`, `mulmod`, and `as_wei_value`

Arguments are now correctly evaluated and cached before being included in the intermediate representation code.

However, builtin functions `ecadd` and `ecmul` are still affected by the same bug. The following code does not compile but should:

```
@external
def foo() -> (uint256[2]):
    a: Foo = Foo(msg.sender)

    return ecmul(a.bar(), 2)

interface Foo:
    def bar() -> uint256[2]: nonpayable
```

5.32 Pointless Assert

Design Low Version 1

CS-VYPER_JANUARY_2023-032

`get_expr_info()` in `vyper.semantics.analysis.utils` contains the assert:

```
assert t is info.typ.get_member(name, node)
```

Since `t` has just been defined as `t = info.typ.get_member(name. node)`, and no mutating operation has occured, the assert will always pass.

5.33 Positional Only Arguments Are Allowed but Ignored in Function Definitions

Design Low Version 1

CS-VYPER_JANUARY_2023-033

Python allows specifying positional only arguments in a function definitons, which are accessible through the `posonlyargs` field of the `arguments` AST node. Since the `arguments` VyperNode does not sets `posonlyargs` as a `_only_empty_fields`, the field can be populated but is ignored.

The following code compiles:

```
@internal
def f(a:uint256,/): #this does not actually defines argument a
            return

@external
def g():
            self.f() #f is called without arguments
```

5.34 RawRevert Should Be Set as Terminal Node

Design Low Version 1

CS-VYPER_JANUARY_2023-034

Builtin `raw_revert` has field `_is_terminus` unset. `_is_terminus` specifies if the node can terminate the branch of a function body which has a non empty return type. The evaluated function is left when `raw_revert` is called, so its `_is_terminus` attribute should be set to `True`, as is the case for `SelfDestruct`.

5.35 Safety Check for Bytestring Classes Not Reacheable

Correctness **Low** **Version 1**

CS-VYPER_JANUARY_2023-035

Function `from_annotation` in class `semantics.types.bytestrings._BytestringT` validates that the bytestring type, such as `String` or `Bytes`, is not being used without a length specifier (`String[5]`). However the function `from_annotation()` of a type is not called in `type_from_annotation()` if the type annotation is an `ast.Name`, so the check at line 126 of `semantics/types/bytestrings.py` is not effective.

5.36 Storage Location of Constants Set to Storage

Design **Low** **Version 1**

CS-VYPER_JANUARY_2023-036

In `visit_VariableDecl` of `vyper.semantics.analysis.module`, the `DataLocation` of constant variables is set to `STORAGE`. While this has no immediate consequences, since constants can't be assigned, it could be misleading and generate problems in future code changes.

5.37 Struct Creation Without Assignment Results in Cryptic Error Message

Correctness **Low** **Version 1**

CS-VYPER_JANUARY_2023-037

The check at lines 512-519 of `vyper/semantics/analysis/local.py` should output an error message when builtins or structs are called without assignment, however the `_id` attribute of `fn_type` is accessed, which causes another exception to be thrown for `TYPE_T(StructT)`, since they have no `._id` field.

Example:

```
struct A:
    a:uint256
@internal
def aaa():
    A({a:1})
```

5.38 Tuple Node Input Does Not Work With Validate_Expected_Type

Correctness Low Version 1

CS-VYPER_JANUARY_2023-038

Function `validate_expected_type` has a branch for the case when `node` is an instance of `vy_ast.Tuple`. However, it is not clear what the purpose of handling `Tuple` nodes is, since the expected type has to be a dynamic or static array.

5.39 VarInfo for `self` Not Constant

Design Low Version 1

CS-VYPER_JANUARY_2023-039

While `self` contains mutable variables, it would make sense that its VarInfo was set as constant. The compilation of the following fails in code generation, while it could fail in type checking.

```
@external
def f():
    self = self
```

6 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

6.1 Comments on PR 2974/3182

Note Version 1

These PR refactor the type system, and unify the front-end and back-end type systems.

6.2 Note on PR 3213: Fix: Constant Type Propagation to Avoid Type Shadowing

Note Version 1

Pull Request 3213 correctly fixes an issue where the type inference for the iterator of for loops would result in validating conflicting types.

Instead of accessing the `"type"` metadata property of a node, which could be dirty with a provisional invalid type, `get_possible_types_from_node()` now accesses the new `"known_type"` metadata property, which is only assigned during constant folding.

6.3 Note on PR 3215: Raise Clearer Exception When Using a yet Undeclared Variable in a Type Annotation

Note Version 1

This PR correctly resolves a cryptic error message cause by undeclared variables during constant folding.

The following line caused problems, when `name` was not in the namespace and `self` had not been declared yet (during constant folding):

```
if name not in self.namespace and name in self.namespace["self"].typ.members:
```

The rhs condition `name in self.namespace["self"].typ.members` would raise an exception when evaluated, because `self.namespace` had no `self` member yet.

Now the condition has been split in 3 conjunctions:

```
if (
    name not in self.namespace
    and "self" in self.namespace
    and name in self.namespace["self"].typ.members
):
```

When `name` is not in `self.namespace`, the condition `"self" in self.namespace` guards against raising accidentally when evaluating the 3rd conjunction.

A more legible exception is raised immediately after the `if` body, when `t = self.namespace[node.id]` is evaluated, since `self.namespace` does not containt `node.id`.

6.4 Note on Pull Request 3194: Fix Raise UNREACHABLE

Note **Version 1**

PR 3194 fixes a bug in code generation that would cause a `raise UNREACHABLE` statement to cause a compiler panic. It is implemented correctly and the resulting code is correct. We noticed that in the generated code a `STOP` unreachable instruction is present after the `INVALID` instruction generated by the `raise` statement.

6.5 Note on Pull Request 3222: Fix: Folding of Bitwise Not Literals

Note **Version 1**

PR 3222 reimplements the binary inversion of literals during folding. The operation computes the 256 bits wide binary inverse, so that the result of the operation should always fits within `uint256`.