PUBLIC

Limited Code Review

of the Sway Semantic Analysis

October 8, 2024

Produced for

by

Contents

1 Executive Summary

Dear Sway Team,

Thank you for trusting us to help you with this limited code review. Our executive summary provides an overview of subjects covered in our review of the latest version of Sway Semantic Analysis according to the Scope.

Limited code reviews are best-effort checks and do not 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. Given the large scope and codebase and the limited time, the findings are not exhaustive.

Fuel implements various semantic analyses on the abstract syntax tree (AST) of Sway programs including the type checking. Their goal is twofold. Firstly they guarantee that the contract to be compiled is semantically correct e.g., functions called are in scope. Secondly, they gather information about the code to facilitate further compilation steps such as the IR generation.

During the review, we were able to identify a high number of issues, some of which of high severity. Most of them relate to the type system. The most important issues involve invalid Sway programs that are accepted by the compiler. Such programs could potentially entail undefined behavior during the execution. More specifically, we uncovered an issue where a big enough function would not be exhaustively checked to return a value from all its possible paths (see Return Path Analysis can be bypassed). Due to incorrect unification of types, a value representing an element of a vector could have a completely different type than what the type of the vector would impose (see Type Propagation in Generics), and arbitrary casting between numeric types would be allowed (see Type Propagation With Numerics). Due to issues in the mutability checks, one could mutate the fields of immutable composite types (see Mutability for immutable composite types) . As Sway aims to implement a type system similar to Rust, we also reported discrepancies between the two languages.

Please note that this limited review is still ongoing.

It is important to note that limited reviews are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. 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	8
• Code Corrected	8
Medium-Severity Findings	8
• Code Corrected	1
• No Response	7
Low-Severity Findings	10
• No Response	10

2 Assessment 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 consisted of a non-exhaustive general review of the semantical analysis in the sway compiler.

This review was not conducted as an exhaustive search for bugs, but rather as a best-effort sanity check.

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	25 June 2024	03c09bf865aa4d4a2855377e5a68c574fa644166	Initial Version
2	24 Sept 2024	f55c81cce61aac31913ac0e87306cbaed7da679a	High-severity fixes
3	8 Oct 2024	66bb430395daf5b8f7205f7b9d8d008e2e812d54	Ongoing review

2.1.1 Included in scope

The scope consists of all following files and directories under `sway/sway-core/src/semantic_analysis/`:

- `ast_node/*`
- `Namespace/*`
- `type_check_analysis.rs`
- `type_check_context.rs`
- `type_check_finalization.rs`
- `coins_analysis.rs`
- `collection_context.rs`
- `module.rs`
- `node_dependencies.rs`

as well as all the files and directories under `sway/sway-core/src/type_system/`.

2.1.2 Excluded from scope

All other files and imports that were not mentioned in Scope. Moreover, bugs related to the Rust compiler itself are considered out-of-scope.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the source as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Fuel implements the semantic analysis for the Sway programming language. Sway is a programming language that targets the FuelVM. The compilation process can be split into the following steps:

1. Source Code: A set of scripts/predicates/libraries/contracts written in Sway and organized in modules.
2. Lexical Analysis: The source code is split into tokens.
3. Syntax Analysis: The tokens are assembled in a concrete syntax tree which is then converted to an abstract syntax tree (AST).
4. **Semantic Analysis**: Various checks on the AST for semantic errors e.g., type checking.
5. Intermediate Representation: The syntax tree is compiled into IR.
6. IR Optimizations: The IR is transformed in various ways to facilitate code generation and improve the efficiency of the final bytecode.
7. Code generation: FuelVM bytecode is produced.
8. Deployment & Execution

The semantic analysis step in the Sway compiler is responsible for verifying the coherence of the code by checking the AST for semantical errors such as type mismatches or undeclared variables. Furthermore, the semantic analysis is in charge of scope resolution, type inference and name resolution. During this step, the compiler transforms the AST, built during the lexical analysis, into a typed AST i.e., an AST where each node is assigned a type. The semantic analysis consists of multiple steps which are described in the following sections.

2.2.1 Module Dependency Graph

The first step in the semantic analysis is to build a module dependency graph for every module. The module dependency graph is a directed graph where each node represents a sub-module and each edge represents a dependency between two sub-modules. The module dependency graph is used to determine the order in which the sub-modules of a module should be compiled. It is important to note that the module dependency graph is a directed acyclic graph (DAG) and that the compiler will fail if a cycle is detected in the graph. This can happen for example in case of circular dependency where two sub-modules depend on each other. All module dependency graphs are built by traversing in a DFS manner the tree formed from the root module and its sub-modules (which themselves can have sub-modules). A module dependency graph is built for every module and its sub-modules and verified to not contain cycles.

2.2.2 Symbol Collection & Namespace Building

The next step is to collect all symbols from the AST and build namespaces for each module. At the time of this review, this step is implemented separately but not used in the compiler pipeline yet. Instead, the symbols are collected and the namespaces are built and filled during the type-checking step.

For each module, the compiler collects all symbols from the AST and builds a namespace for the module. The namespace contains a tree of lexical scopes corresponding to the scopes in the module. Each lexical scope is a map from a symbol identifier to a declaration.

2.2.3 Type Checking

The type-checking step is responsible for verifying that the types of expressions in the AST are correct and converting the AST into a typed AST. Type checking starts at the root of AST by invoking `tyProgram::type_check` which in turn calls `tyModule::type_check` for the root module. All of the module's sub-modules are then recursively type-checked. A module consists of a vector of `AstNode` which is first ordered by dependencies by calling `node_dependencies::order_ast_nodes_by_dependency`. Then the ordered AST nodes are

type-checked in `TYModule::type_check_nodes()` by calling `TyAstNode::type_check` for each one of them.

An AST node can represent one of the following types:

- `UseStatement`: a `use` statement allowing to bring symbols from another module into the current module's scope.
- `IncludeStatement`: a `mod` statement that includes another module into the current module.
- `Declaration`: a declaration of a variable, a function, a type, a trait, etc.
- `Expression`: an expression such as a binary operation, a function call, an `if` statement, etc.
- `Error`: a malformed expression that violates the language syntax

`UseStatement`
The typing of a `UseStatement` is done by building a `TyAstNodeContent::SideEffect` containing `TySideEffectVariant::UseStatement` holding the import type, alias, call path and span of the statement. During the type checking the namespace of the module is expanded with the imported symbol.

`IncludeStatement`
The typing of an `IncludeStatement` is done by building a `TyAstNodeContent::SideEffect` containing a `TySideEffectVariant::IncludeStatement` holding the name of the imported module, the span and the visibility of the statement. The module's namespace is not modified.

`Declaration`
The typing of a `Declaration` is done by building a `TyAstNodeContent::Declaration`. The different declaration types are :

- `VariableDeclaration`: resolves the type of the ascription (if any), type checks the RHS expression and verifies that the type of the RHS expression is coherent with the ascription.
- `ConstantDeclaration`: resolves the type of the ascription (if any), type checks the RHS expression against the ascription and verifies that the type of the RHS expression is coherent with the ascription.
- `ConfigurableDeclaration`: resolves the type of the ascription (if any), type checks the RHS expression against the ascription and verifies that the type of the RHS expression is coherent with the ascription.
- `TraitTypeDeclaration`: resolves the type of the declared type in the trait type declaration.
- `EnumDeclaration`: Type checks the enum declaration by first creating a new namespace to create a scope for generic type parameters. Then, it type checks the type parameters followed by all the enum variants.
- `EnumVariantDeclaration`: part of the enum declaration type checking, type checks every enum variant expression.
- `FunctionDeclaration`: Type checks the function signature by checking the function parameters and then the return type. A new namespace is created for the function scope. Then the function body is type-checked to a `TyCodeBlock` using a new namespace initialized with the previously type-checked type and function parameters. Finally, a check is performed to ensure that the function type parameters implement the required traits.
- `TraitDeclaration`: Type checks a trait declaration by first type-checking the trait and then each of its super traits.
- `ImplTrait`: Type-checks an `impl` block of a trait for a certain "target" type, by first type-checking the generic type parameters (if any), then unifying the trait's `Self` type with the concrete "target" type, performing the appropriate monomorphizations, and finally adding the resulting interface surface to the appropriate scope.

- `ImplSelf`: Mostly as above, but refers to an `impl` block for a type, regardless of traits.
- `StructDeclaration`: Type checks a struct declaration by creating a namespace for the declaration and type-checking each field of the struct.
- `AbiDeclaration`: Type checks and adds in scope every item of an ABI declaration (i.e. contract interfaces), with generics disallowed, and recursively type-checks all the ABI supertraits.
- `StorageDeclaration`: Type checks a storage declaration in a Sway contract by iterating over every storage entry and type-checking it.
- `TypeAliasDeclaration`: Type checks an aliased type declaration by resolving the type being aliased and then creating a typed alias declaration.

All declarations update the module's namespace by inserting the declaration into it.

`Expression`
The typing of an expression is done by building a `TyAstNodeContent::Expression`. The different expression types are :

- `Literal`: Type checks a literal (e.g., a `Numeric`, a `String`, a `Boolean`) by converting it to a typed literal (e.g respectively a `TypeInfo::Numeric`, a `TypeInfo::StringSlice`, a `TypeInfo::Boolean`). The type is then inserted into the type engine and a typed literal expression is returned.
- `AmbiguousVariableExpression`: Resolves the ambiguity of the variable expression by either type-checking the ambiguous variable expression to a typed delineated path or a typed variable expression. The ambiguity is resolved by checking if the variable is an enum variant or not.
- `Variable`: type checks a variable expression by looking up the variable declaration in the namespace and returning a typed variable expression. The variable's type can either be a variable, constant, configurable or an abi declaration.
- `FunctionApplication`: Type checks a function application by first retrieving the function declaration from the namespace and then type-checking the function arguments and the return type. The function application typing also checks the arity of the function arguments, purity of the callee and caller, and checks trait constraints that might apply.
- `LazyOperator`: Type checks lazy operators (`||` and `&&`) by type-checking the left and right operands and returning a typed lazy operator expression.
- `CodeBlock`: Type checks a code block by typing all AST nodes inside the code block and then type checking the return type of the code block. If the block contains either a `Return`, `Break` or `Continue` expression or one of the AST nodes has `Never` as the return type, the block's return type is `Never`.
- `If`: Type checks an `if` expression by type-checking the condition to be boolean, and then type-checks both the "then" and "else" code blocks; a non-existent "else" branch is taken to be of type unit. The two types are then unified among them, and with the expectation (e.g. an ascription) coming from the calling context.
- `Match`: A match expression is typed by first type checking the expression matched on. Then, the match arms are type-checked and a `TyMatchExpression` is built. Furthermore, the match arms are checked to be exhaustive and reachable by computing the usefulness of each arm.
- `Asm`: checks that no control flow opcodes are used and ensures that initialized registers are not reassigned. Type checks the asm block by type-checking the registers and the return type of the asm block.
- `Struct`: Type checks a struct expression by type-checking each field of the struct. The types of the struct fields and the fields in the definition of the struct are unified. Furthermore, checks are performed to ensure that no extra fields are present, that all required fields are present, and that the struct visibility is public if it is imported from a sub-module.

- `Subfield`: represents an access to a subfield of the form A.B where B is a field of a struct A. The type checking first initializes this access to type `Unknown` and then recursively unwraps the subfield access until the accessed field is not a struct anymore. (This is done to support A.B.C.D accesses). Then, the field is type-checked and verified to be accessible from the current scope.
- `MethodApplication`: Type checks a method application, for example `a + b`. First, the function arguments are type-checked. Then, the method is monomorphized using the typed arguments followed by a unification step to unify the return type with the annotated return type of the method. The function arguments are then type-checked a second time. Furthermore, orthogonal checks to type checking are performed to ensure that the method's purity of the callee and caller are the same and that the method is visible in the current scope. Additionally, if the method is non-payable, but is called with some possibly non-zero coins (determined through a heuristic), an error is raised. Finally, if the method is of the type `a.b(c)` the first argument is checked to be `self`.
- `Tuple`: Type checks a tuple by type-checking each element of the tuple and returning a typed tuple expression containing the typed elements.
- `TupleIndex`: Type checks an access to a tuple element of the form `t.x` where `x` is an integer literal. First, it creates a context where the access is type-checked to `Unknown`. Then, the prefix `t` is type checked followed by the type checking of the access. Furthermore, the access index is verified to be within the bounds of the tuple.
- `AmbiguousPathExpression`: an expression for which the parser did not manage to decide whether it is a free function call, an enum variant, or a UFCS-style method call. This ambiguity is resolved now, at type-checking time, when more context information is available.
- `DelineatedPath`: First, determines if the call path is from a module, enum, function or constant. If more than one reference is found among all options listed before, the path is ambiguous and an error is raised. If the path is not ambiguous, the path is type-checked to the correct expression type.
- `AbiCast`: Type checks the contract address to ensure that it is a valid contract address. Then retrieves the abi declaration from the namespace and retrieves the abi items from it. Then, the interface surfaces and methods of super traits implemented by that abi are recursively made available to the abi cast expression by adding them to the namespace as trait implementations.
- `Array`: Type checks an array declaration by first determining if the array is empty or not. If it is, the type `Never` and a length of 0 is returned as part of the typed array. If not, the initial type of the array is determined by the type ascription if it exists, else it is typed to `Unknown`. Then, every element of the array is type-checked against the initial type of the array.
- `ArrayIndex`: Type checks an array index access of the form `a[i]` by first type-checking the prefix (a). The typing of the prefix is recursive as an access can be of the form `a[i][j]` in the case of nested arrays. Then, the index is type-checked to be of type `UnsignedInteger(IntegerBits::SixtyFour)`.
- `IntrinsicFunction`: type-checks an application of a compiler-intrinsic function, such as `log()` or `state_load_quad()`.
- `WhileLoop`: Type checks a while loop expression by first type-checking the condition to be a boolean. The return type of the body is forced to be `Unit`: a while loop cannot take up a value. Then, the body (a `CodeBlock`) is type-checked.
- `ForLoop`: Type-checks the desugared version of the `for` loop. During the parsing stage, a `for` loop is desugared into a codeblock containing the declaration of the loop iterable, and `while true` containing the exit condition on the iterator (with a `break`) followed by the actual content.
- `Break`: Gets type checked to `Never`.
- `Continue`: Gets type checked to `Never`.
- `Reassignement`: Type checks an assignment, either to a variable (which needs to have been declared as mutable), or an element of a complex type, like a struct or array. In either case, type

checking and unification is carried out against the (possibly partial) information on the expected type of the RHS.

- `ImplicitReturn`: Type checks an implicit return by type-checking the expression to be returned. The returned type of the implicit return is the type of the expression. The parser maintains the invariant that only one implicit return can be present per function/block.
- `Return`: Type checks to `Never` but itself contains a typed expression that is returned.
- `Ref`: Type checks a reference expression by first using the type annotation if it exists. Then, the referenced expression is type-checked and a typed reference expression is returned.
- `Deref`: Type checks a dereference expression by first type-checking the expression to be dereferenced. Then, the dereferenced expression is returned.

Once an expression has been type-checked, the type returned by the expression is unified with the type annotation of the expression if it exists. If the expression type cannot be cast to the type annotation, a type error is raised.

After generating a typed program, the compiler checks that there are no recursive function calls in the program as there is no support yet for them. If a recursive function call is found, the compiler will raise an error.

2.2.4 Type unification

Type unification is a sub-process of type inference, whereby some newly-obtained context information (an ascription, an assignment, a usage as a function parameter) allows to further restrict the typing of a partially-typed item, in a fashion akin to set intersection. The outcome might still be a partially-resolved type (e.g. unifying `(_, u8, _)` with `(str, _, _)` yields `(str, u8, _)`), a fully-resolved concrete type (e.g. unifying `(u8, _)` with `(_, u32)` yields `(u8, u32)`), or the empty set, leading to a type-mismatch compiler error (e.g. unifying `bool` with `str`).
In Sway, this is implemented by roughly having each item in the program (variables, struct fields, function arguments) have its own unique `TypeId`, which is just a key into a compiler-wide dictionary, mapping each `TypeId` to a `TypeInfo`, a representation of the current typing for that item. In a few cases where two items are found to be of the exact same type (e.g. a direct assignment), one of the `TypeId` gets substituted with the other: this way, when unification happens on one of the items, and the `TypeInfo` corresponding to that `TypeId` gets updated, the update will automatically reflect on the other item as well.
Unification itself roughly works by substituting the less restrictive of the two `TypeInfo` with the more restrictive, if there is an intersection. This happens field-by-field for composite types like structs and tuples.

2.2.5 Control Flow Analysis

After the creation of the typed AST, a control flow graph (CFG) is built to perform a control flow analysis on the program. First, it checks if there is any dead code which is detected by finding a cluster of one or more orphan nodes in the CFG. If there is dead code, a warning is emitted. Then, return path analysis is performed to ensure that all paths in the CFG return a value of the correct type. If a path returns a value of the incorrect type, an error is raised.

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 any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- 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	0
Medium-Severity Findings	7

- Additional Constraints for Generics Are Not Respected
- Constraints as Hints
- Implicit Casting of Array Elements
- Implicit Trait Constraints
- Propagation From Type Parameters
- Trait Constraints Are Not Respected
- Trait Functions With the Same Name Get Disambiguated Arbitrarily

Low-Severity Findings	10

- Cannot Use Generic Trait in Trait Constraints
- Dangling Generics
- Generics for Structs and Functions
- Incomplete Monomorphisation Leads to ICE
- Module Dependencies Can Be Cyclic
- Never Type For Arrays
- No Warning for Overlaps in Pattern Matching
- Parsing Generics
- Type Constructors as Type Arguments
- Unification of Never Type

5.1 Additional Constraints for Generics Are Not Respected

Correctness Medium Version 1

CS-FSSA-001

Sway allows users to set trait constraints on specific items of a struct. Take the following example:

```
trait T1 {}
trait T2 {}
```

```
struct S<T> where T: T1 {
  x: T,
}

impl<T> S<T> where T: T1{
  fn only_t1(self) {
     Self::also_t2(self);
  }

  fn also_t2(self) where T: T2{
     log("Hello");
  }
}

impl T1 for u8 {}

impl T1 for u64 {}
impl T2 for u64 {}

fn main() {
  let a = S::<u8>{x: 42};
  a.only_t1(); //prints Hello but u8 doesn't implement T2
}
```

When implementing the generic `S<T>` we can define `also_t2()` under the additional condition that `T` implements `T2` besides `T1`. Note, however, that we are able to call `also_t2()` from `only_t1()` even though the added constraint is not necessarily satisfied for all of its possible type parameters (e.g. `u8`). The program executes normally in such a case even though it should be rejected.

Taking the previous bug a step further, the following code also type checks but the compiler later encounters an internal error and fails as it cannot resolve `t2()` for `u8` as it doesn't implement the `T2`.

```
script;

trait T1 {
   fn t2(self);
}

trait T2 {
   fn t2(self);
}

struct S<T>
   where T: T1,
{
   x: T,
}

impl<T> S<T> where T: T1 {
   fn only_t1(self) {
       Self::also_t2(self);
   }

   fn also_t2(self) where T: T2{
```

```
        self.x.t2();
    }
}

impl T1 for u8 {
    fn t2(self){
        log("t18");
    }
}

impl T1 for u64 {
    fn t2(self){
        log("t164");
    }
}

impl T2 for u64 {
    fn t2(self){
        log("64");
    }
}

fn main() {
    let a = S::<u64>{x: 42};
    a.only_t1();
}
```

The error message we get is:

> Trying to compile a deferred function application with deferred monomorphization. Please file an issue on the repository and include the code that triggered this error.

5.2 Constraints as Hints

Correctness **Medium** **Version 1**

CS-FSSA-002

Generic structs can impose constrants on their generic types. In the following example, we define S<T> but we require `T` to implement `MyTrait`:

```
trait MyTrait{}

struct S<T> where T: MyTrait{
    x: T
}

impl MyTrait for u8 {
}

fn main() {
  let s: S<_> = S{x: 42};
}
```

We implement the trait for u8. When we instantiate s, the compiler should infer its type as S<u8> since u8 is the only type implementing `MyTrait` (this is Rust's behaviour); alternatively, it might decide not to

have enough information to carry out type inference on s, and refuse compilaiton for that reason. Instead, the compiler rejects the program because it wrongly assumes that the generic parameter is u64.

5.3 Implicit Casting of Array Elements

Correctness Medium Version 1

CS-FSSA-003

When it comes to typing the elements of an array, the type system is not able to stick to proper types. This leads to the following snippets all successfully compiling:

```
let a = [8, 8, 8u64];
let b: u32 = a[2];
```

```
// compiles even though we have given a hint
let a = [1u32, 1u64];
let b:u32 = a[1];
```

```
let mut a = [8u16, 8u16, 18446744073709551615];
log(a[2]); // Logs 2**16 - 1
```

Note, this issue relates to two other issue reported regarding typing of arrays and typing of literals.

5.4 Implicit Trait Constraints

Correctness Medium Version 1

CS-FSSA-004

In Sway, traits can define supertraits. This means that, to implement such a trait for a type, one must implement its supertrait as well. For example, T2 is a supertrait of T1:

```
script;

trait T2 {}

trait T1: T2 {
    fn new() -> Self;
}

struct S {}

impl T2 for S {}

impl T1 for S {
    fn new() -> Self {
        S {}
    }
}

fn bar<T>() -> T
where
    T: T1,
```

```
{
    T::new()
}

fn foo<T>() -> T
where
    T: T2,
{
    bar()
}

fn main() {}
```

In the above example, let's focus on `foo()`. `foo()` is a generic function, whose type parameter `T` just has to implement `T2`. In its implementation, `foo()` calls `bar()` and returns its result: type inference therefore implies that the same type `T` should also parametrise the call to `bar()`. However, `foo()` additionally requires that its type parameter implement `T1`: this constraint is not automatically satisfied for all types `T` that implement `T2`, so compilation should fail. Instead, the program is accepted by the compiler.
Taking this a step further, if one adds `let s: S = foo();` to the `main()`, so as to actually trigger a call to a non-existent function, the compiler runs into an ICE:

> Internal compiler error: Method new_3 is a trait method dummy and was not properly replaced. Please file an issue on the repository and include the code that triggered this error.

For reference, Rust prohibits the above snippet and requires to add an explicit trait bound `T1` for `foo()`.

5.5 Propagation From Type Parameters

Correctness **Medium** **Version 1**

CS-FSSA-005

Sway's type system ignores some hints about the type of variables. Let us consider the following example:

```
trait Build {
    fn build() -> Self;
}

fn produce<T>() -> T
where
    T: Build,
{
    T::build()
}

impl Build for u32 {
    fn build() -> Self {
        31
    }
}

impl Build for u64 {
    fn build() -> Self {
```

```
        63
    }
}

fn consume(a: u32) {}

fn main() {
    let x = produce();
    consume(x);
}
```

Here we instantiate variable `x` with the result of `produce()`. Since `produce()` returns a generic type, the concrete type of `x` cannot be determined just yet: it can either be `u64` or `u32`. Since `x` is later passed to `consume()`, which only accepts `u32`, type inference should assign this type to `x`. However, the compiler fails to use this information and rejects the program, as it assumes `x` to be of type `u64`.

5.6 Trait Constraints Are Not Respected

Correctness **Medium** **Version 1**

CS-FSSA-006

In Sway similarly to Rust, one can define function over a generic type which can be further constrained by a trait. An example can be found in the following snippet:

```
trait MyTrait1{
    fn foo();
}

fn bar<T>() -> Option<T> where T: MyTrait1{
  None
}
```

Here `bar`, returns a generic Option `T` where `T` should implement `MyTrait`. When we force the result to monomorphize with a type ascription, the compiler should check if the concrete type satisfies the trait bounds. However, this doesn't happen therefore the example below successfully compiles.

```
fn main(){
  let x: Option<u32> = bar();
}
```

Note, however, that the following, seemingly equivalent snippet, will be rejected by the compiler:

```
fn main(){
  let x = bar::<Option<u32>>();
}
```

5.7 Trait Functions With the Same Name Get Disambiguated Arbitrarily

Correctness **Medium** **Version 1**

A type can implement two different traits that define the same function. A UFCS-style call syntax is then needed to explicitly disambiguate because calls like `x.foo()` will raise a compiler error.
Such an error, however, is not raised if the type is wrapped in a generic struct that has two generic `impl` blocks, one bound to the first trait, one to the other, as in the following code snippet:

```
script;

trait Cat {
    fn speak(self) -> u64;
}
trait Dog {
    fn speak(self) -> u64;
}

struct S<T> {
    x: T,
}


impl<T> S<T>
where
    T: Cat,
{
    fn foo(self) -> u64 {
        self.x.speak()
    }
}

impl<T> S<T>
where
    T: Dog,
{
    fn foo(self) -> u64 {
        self.x.speak()
    }
}



impl Dog for u64 {
    fn speak(self) -> u64 {
    log("2");
        2
    }
}

impl Cat for u64 {
    fn speak(self) -> u64 {
    log("1");
        1
    }
}

fn main() {
```

```
    let s = S::<u64> { x: 1 };
    s.foo();
}
```

The compilation succeeds and the execution output shows that it is the first generic `impl` (the one bound to `Cat`) that prevails.
To the contrary, Rust refuses the compilation of an analogous snippet.

5.8 Cannot Use Generic Trait in Trait Constraints

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

CS-FSSA-008

The same code snippet as in Incomplete monomorphisation leads to ICE cannot be made to compile, even if the call to `feed()` is replaced with its fully-qualified version `feed::<Brain, Zombie>()`.

The compiler raises the following error:

```
feed::<Brain, Zombie>(b);
^^^^^^^^^^^^^^^^^^^^ Trait "Devour<T>" is not implemented for type "Zombie".
```

The compiler seems to interpret the trait constraint `where U: Devour<T>` "literally", without realizing that `T` is a generic.

5.9 Dangling Generics

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

CS-FSSA-009

Similarly to Rust, Sway offes generics. Generic values should be eventually be resolved by the type system. Unlike Rust, Sway still accepts programs with dangling generics like the following:

```
fn bar<T>() -> Option<T> {
  None
}

fn main(){
  bar();
}
```

Not that in the example above, the compiler cannot know from the context what is the concrete value of `T` to monorphize bar.

5.10 Generics for Structs and Functions

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

CS-FSSA-010

Users can define generic structs that implement generic functions. For example the following is a valid Sway program:

```
script;

struct S<T> {
    x: T,
}

struct M1 {}

impl<T> S<T>
{
    fn bar<M>()
    {
    }
}

fn main() {
    let x = S::<M1>::bar::<M1>();
}
```

However, if we introduce trait constraints, the compilation fails:

```
script;

trait MyTrait {
}

struct S<T> {
    x: T,
}

struct M1 {}

impl MyTrait for M1 {
}

impl<T> S<T>
where
    T: MyTrait,
{
    fn bar<M>()
    {
    }
}

fn main() {
    let x = S::<M1>::bar::<M1>();
}
```

5.11 Incomplete Monomorphisation Leads to ICE

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

CS-FSSA-011

Sway monomorphises generic functions, based on the concrete types provided in each application; however, these types might need to be inferred from some context information. When such information is not sufficient to fully determine the concrete types, the compiler should raise an error and fail.
Instead, the following code snippet triggers an Internal Compiler Error:

```
script;

trait Devour<T> {
    fn eat(t: T);
}

struct Brain {
    grams:  u64,
}

struct Zombie {
    name:   str,
}

impl Devour<Brain> for Zombie {
    fn eat(b: Brain) {
    }
}

fn feed<T, U>(t: T)
    where U: Devour<T>,
{
    U::eat(t);
}

fn main() {
    let b = Brain{grams: 250};

    feed(b);
}
```

The compiler raises the following ICE:

```
fn eat(t: T);
   ^^^ Internal compiler error: Method eat_2 is a trait method dummy and was not properly replaced.
Please file an issue on the repository and include the code that triggered this error.
```

This probably indicates that the compiler did not detect the impossibility of carrying out type inference, and instead moved on to a later stage where functions are expected to be fully monomorphised.

5.12 Module Dependencies Can Be Cyclic

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

CS-FSSA-012

An early check in the compilation process, before type-checking begins, ensures that the module dependency graph has no cycles. However, the algorithm employed is broken, as it only goes "one level deep", instead of building the full dependency graph: for each (sub)module, it only looks for dependency cycles among the submodules directly imported (plus the "parent" module itself).
As a result, in a project structure like the following:

```
src/
|---main.sw
|---a.sw
|---a/
    |---b.sw
    |---b/
        |---c.sw
```

one can craft the dependency cycle `main -> a -> b -> c -> a`.

5.13 Never Type For Arrays

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

CS-FSSA-013

Empty arrays are assigned the never type. Never type is treated as a subtype of all types. This means that a never array should be assignable to an empty array of any other type. Moroever, if we assign an empty array to a type-ascribed variable, the type of this variable should be its ascription. This is not true in the following example.

```
script;

impl [u32;0] {
  fn foo(self){
      log("32");
    }
}

impl [!;0] {
  fn foo(self){
      log("never");
    }
}

fn main() {

  let x:[u32;0] = [];
  x.foo(); // logs "never"
}
```

The never type in arrays is more problematic considering the following case which also compiles:

```
let x:[u32;0] = [];
let y: [!;0] = x;
```

As one would expect `x` to preserve its ascription assigning `x` to `y` should fail since the never type is a subtype of the `[u32;0]`.

5.14 No Warning for Overlaps in Pattern Matching

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

CS-FSSA-014

During pattern-matching a user can define a disjunction of multiple different values as a scrutinee for a particular arm like the following example:

```
match v {
 1 =>  …
 1 | 3 => …
}
```

Such a case will not yield any warning however. The root cause is the initialisation of `witness_report` at the beginning of `usefulness.rs::is_useful_or()`. It is initialised to an empty `Witnesses` (which still counts as true) instead of `NoWitness`, as is the case for e.g., `is_useful_constructed()`. Therefore, `is_useful_or()` will essentially always return true.

5.15 Parsing Generics

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

CS-FSSA-015

Sway supports generic types. The concrete type is specified as a suffix for the generic type. However, the following parses successfully:

```
struct S<T> {
    x: T,
}

fn main() {
    let x: <u8>::S::<u8> = S::<u8>{x: 8};
}
```

Note that the prefix type is completely ignored. However, such program should be rejected by the compiler.

5.16 Type Constructors as Type Arguments

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

CS-FSSA-016

In Sway one can define generics like the following:

```
struct S<T>  {x: T}
```

The programmer then needs to provide a concrete type to instantiate the struct.
However, the following is wrongly accepted by Sway:

```
struct S<T> {
    x: T,
}

impl<T> S<T>
{
    fn bar(){}
}
```

```
fn main() {
    let x = S::<S>::bar();
}
```

Note that here S does not represent a type but rather a type constructor i.e., an element of the language that accepts a concrete type T and returns another concrete type S<T>. This shouldn't be accepted.

5.17 Unification of `Never` Type

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

CS-FSSA-017

Some operators like `return` are assigned to a `Never` type also denoted with `!`. This represents expressions whose evaluation cannot be assigned if they're on the right-hand-side. When never is unifified with another type, then the other type should dominate. Consider the following example:

```
fn main(){
  let mut v: u32 = 1; // (1)
  v = return; // (2)
}
```

In the example, the type of `v` is `u32` from `(1)` and is also assigned a `!` from `(2)`. Since `!` is always unified with any type the snippet successfully type checks. However, the following does not type check, even though it's a similar case.

```
fn main(){
  let mut v: u32 = return;
  v = 1;
}
```

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

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

Critical-Severity Findings	0
High-Severity Findings	8

- Exhaustiveness Check Against Configurables Code Corrected
- Mutability for Immutable Composite Types Code Corrected
- Return Path Analysis Can Be Bypassed Code Corrected
- Termination of Return Path Analysis Code Corrected
- Type Checking of Arrays Code Corrected
- Type Propagation With Numerics Code Corrected
- Type Propagation in Generics Code Corrected
- Typing Numeric Literals Code Corrected

Medium-Severity Findings	1

- Multipe Field Initializations Code Corrected

Low-Severity Findings	0
Informational Findings	1

- Documentation Link in usefulness.rs Does Not Refer to a Valid Location Code Corrected

6.1 Exhaustiveness Check Against Configurables

Correctness High Version 1 Code Corrected

CS-FSSA-028

Sway contracts can define configurable variables. These are constants with a default value that can be changed on deployment. Sway implements pattern matching. During the execution of a pattern match, a value is checked against some scrutinees. As soon as the value matches a scrutinee the particular arm is executed. Pattern matching should be exhaustive i.e., it should cover all possible values a type can take. Exhaustiveness is checked during compile time. Sway allows configurables to be used as scrutinees. For example, the following expression is valid:

```
configurables {
  A = true
}

...

match v {
      A => ...
```

```
        false => ...
}
```

The above snippet can lead to undefined behavior. Consider the case where the deployer of a contract sets A to `false`. In this case the pattern matching is not exhaustive any longer and if the value of v is true it will never be matched.

Code corrected:

Configurables are not allowed to be used as matching arm. An explanatory error message is printed if such a case is detected.

6.2 Mutability for Immutable Composite Types

Correctness | High | Version 1 | Code Corrected

CS-FSSA-030

In Sway, mutability is specified upon declaration. In case a variable is defined as immutable, its content cannot be changed. However, the mutability check is not recursive on composite types. This means we can create a mutable reference to a field of an immutable variable like the following example:

```
let t : (u64, u64) = (17, 42);
let p : &mut u64 = &mut t.0;
*p = 18;
```

Code corrected:

A deep mutability check is now performed.

6.3 Return Path Analysis Can Be Bypassed

Correctness | High | Version 1 | Code Corrected

CS-FSSA-024

At the end of the semantic analysis, a return path analysis is performed to ensure that all paths that are required to return a value do so. This analysis is performed on a return path graph generated for each function. The function `ensure_all_paths_reach_exit` performs this check by traversing the graph node by node and making sure that all end nodes with no neighbors reach the exit node or the return type of the function is unit in which case the function does not return a value.

However, if the graph has a depth of more than 50, the return path analysis will always succeed. This is because a fixed depth of 50 is used to limit the traversal of the graph.

The following code snippet shows how an invalid function can be compiled by bypassing the return path analysis check :

```
script;

fn main(){
}
```

```
fn f() -> bool {

    if true {
        return true;
    } else {
        return false;
    };

    // Repeat the above block 50 times

    // Now anything can be returned here.
}
```

The absence of this check can lead the execution to undefined behavior.

Code corrected:

The traversal limit has been removed.

6.4 Termination of Return Path Analysis

Correctness | High | Version 1 | Code Corrected

CS-FSSA-027

The return path analysis terminates when `rovers[0]` is the exit node. However, this condition doesn't guarantee that the are no other rovers which haven't been visited yet. In such a case, it is likely that the analysis does not visit the neighbors of these rovers, and therefore not all nodes are going to be checked.

Code corrected:

The analysis now terminates only if and only if all nodes of the function have been visited. Moreover, it makes sure not to cover nodes that correspond to inner function declarations like in the following example:

```
fn main() -> u64 {
 impl core::ops::Eq for X {
     fn eq(self, other: Self) -> bool {
         asm(r1: self, r2: other, r3) {
             eq r3 r2 r1;
             r3: bool
         }
     }
 }
 if X::Y(true) == X::Y(true) {
     a
 } else {
     a
 }
```

}

6.5 Type Checking of Arrays

Correctness | High | Version 1 | Code Corrected

CS-FSSA-029

In Sway, the type of an array is inferred using its first element. However no checks are performed later to check if all the elements of the array are of the same type. This means that the following successfully type checks:

```
fn main() {
  let a = [1, 2, "hello"];
}
```

Fortunately, the snippet above doesn't compile since the IR-generation fails. However, the type system is the one responsible for uncovering such issues and therefore, this a very important issue as far as the type system is concerned.

Code corrected:

All the elements of the array are now type-checked.

6.6 Type Propagation With Numerics

Correctness | High | Version 1 | Code Corrected

CS-FSSA-021

Variables in Sway are strongly typed i.e., there cannot be any implicit changes in their type. Consider the following example:

```
script;

trait MyTrait{
  fn foo(ref mut self);
}

impl MyTrait for u64 {
    fn foo(ref mut self) {
      let x: &mut u64 = &mut self;
      log("64");
    }
}

impl MyTrait for u32 {
    fn foo(ref mut self) {
      let x: &mut u32 = &mut self;
      log("32");
    }
}

impl u64 {
    fn baz(self) {
```

```
      log("64");
    }
}

impl u32 {
    fn baz(self) {
      log("32");
    }
}


fn bar<T>(ref mut x: T) where T: MyTrait {
  x.foo();
}

fn main() {
  let mut a = 0;
  bar(a);
  let x: &u32 = &a;
  a.baz();
}
```

In the example, we define `MyTrait` which we then implement for `u32` and `u64`. Moreover, we implement function `baz` for these types. In `main()` we declare variable `a`. Since `a` is assigned a literal with no type ascription, it is initially typed as `numeric`. We execute `bar(a)`. The type checker at this point will resolve to `bar::<u64>` and eventually calls `foo()` for `u64` which logs `64`. This means that `a` should be typed as `u64`. However, we later assign a reference of `a` to `x` which is explicitly typed as `&u32` which coerces `a` to `u32`. We later, call `a.baz()` which calls `foo()` for `u32` and logs `32`. This script is successfully compiled and executed as described, implying a type-propagation failure.

Code corrected:

The type propagation now works as expected. In the example above, 32 is going to be logged.

6.7 Type Propagation in Generics

Correctness **High** **Version 1** **Code Corrected**

CS-FSSA-022

When the type inference meets a generic type with a placeholder it creates an unknown generic type. Normally, when later more infomration is found the type should be updated. However, this is not the case for the following example:

```
script;

trait Wrap {
    fn wrap(self) -> Vec<Self>;
}

impl<T> Wrap for Vec<T> {
    fn wrap(self) -> Vec<Self> {
      let mut v = Vec::new();
      v.push(self);
```

```
      v
    }
}

fn foo() {
    let mut a: Vec<_> = Vec::new();
    let b: Vec<Vec<_>> = a.wrap();
    let c: Vec<Vec<Vec<u8>>> = b.wrap();
    a.push("str");
}

fn main() {}
```

Here the types, and more specifically the unique type ids assigned to values a, b and c respectively aren't properly updated due to the way unification is implemented. As a result, the compiler is not able to deduce that a cannot be a vector of strings if c is a `Vec<Vec<Vec<u8>>>`.

Code corrected:

The type propagation now works as expected. The example above fails to compile and the compiler is able to determine the correct type of the inner most array.

6.8 Typing Numeric Literals

Correctness **High** **Version 1** **Code Corrected**

CS-FSSA-023

In Sway, the following declaration properly fails since a is declared as u32 but is assigned a value that exceeds the size of the type.

```
let mut a: u32 = 18446744073709551615; //2^64-1
```

However, the following case will not fail but perform an implicit typecast instead:

```
fn main() {
  let mut a = 18446744073709551615; //2^64-1
  let x: u32 = a;
  log(x);
}
```

The reason is that a is simply typed as a `numeric` in the absence of a type ascription, i.e., a type that represents either u32 or u64. In this case, the compiler cannot reason about the size of the variable. Later, x is typed as a u32 and therefore a is typed as u32 as well but has a way bigger value assigned to it. This might lead to spurious write's into neighbouring memory areas.

Code corrected:

The numerics are properly checked to not overflow.

6.9 Multipe Field Initializations

Correctness | Medium | Version 1 | Code Corrected

CS-FSSA-026

In Sway a struct can be defined and instantiated like the following example:

```
struct S{x: u32}
...
let s = S{x:1};
```

However the compiler does not reject multiple initializations of the same field. The second time a field is initialized, it's completely ignored and not even type-checked therefore the following instantiation is accepted:

```
let s = S{x:1, x: "a"};
```

Code corrected

The compiler now rejects multiple initializations of the same field, with an explanatory error message.

6.10 Documentation Link in `usefulness.rs` Does Not Refer to a Valid Location

Informational | Version 1 | Code Corrected

CS-FSSA-025

In the file `usefulness.rs` the documentation link on line 23 does not refer to a valid web location.

```
/// Implemented in Rust here:
/// https://doc.rust-lang.org/nightly/nightly-rustc/rustc_mir_build/thir/pattern/usefulness/index.html
```

Code corrected:

The link has been changed to /// https://doc.rust-lang.org/1.75.0/nightly-rustc/rustc_mir_build/thir/pattern/usefulness/index.html

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Out-of-bounds Inconsistency

Informational **Version 1**

CS-FSSA-018

Developers can specify arrays of static size in Sway. The size of these arrays cannot be changed in runtime. This means that the compiler can detect out-of-bounds accesses and writes to these arrays. However, the compiler will not detect out-of-bounds writes to the arrays, in case the value written is of Never type.

```
// This doesn't compile
fn main() {
  let mut a = [];
  let x: u32 = if (true){
    a[0]
  } else {
    42
  }
}

// This compiles
fn main() {approach
  let mut a = [];
  a[0] = return;
}
```

7.2 Redundant Checks for Struct Equality

Informational **Version 1**

CS-FSSA-019

In `unifier.rs::unify_structs()`, of the three joint checks to determine whether the two structs are the same, the first alone would be sufficient: `if rn == en`

7.3 Wrong Error Message Emitted

Informational **Version 1**

CS-FSSA-020

The following snippet emits a wrong/confusing error message, because a `for` loop only gets type-checked in its desugared form, as a `while` loop:

```
fn main() {
  let mut v : Vec<u8> = Vec::new();
```

```
  v.push(1);
  v.push(2);
  v.push(3);

  for elem in v.iter() {
    log(elem);
    3
  };
}
```

The error message is the following:

```
A while loop's loop body cannot implicitly return a value.
Try assigning it to a mutable variable declared outside of the loop instead.
```

8 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.

8.1 Confusing Use of `self`

Note **Version 1**

Sway implements many features similar to Rust. One of them is the use of `Self` to refer to the type of the struct or trait currently implemented. Another one is the use of `self` which refers to an instance of the currently implemented type. It is important to note, contrary to Rust, that the terms `self` and `Self` are used interchangeably in Sway. Consider the following example:

```
script;

trait MyTrait {
    fn foo(self, other: Self) -> Self;
}

impl MyTrait for u8 {
    fn foo(self, other: Self) -> self {
        if self > other {
            self
        } else {
            other
        }
    }
}

fn main() -> () {
    let a = 1u8;
    let b = a.foo(2u8);
    log(b);
}
```

In the example, in the implementation of `MyTrait` for `u8`, `self` is returned instead of `Self`. However, `self` in this case simply refers to the type and not to the value.

8.2 Dereferencing Arrays Is Not Possible

Note **Version 1**

In Sway, developers can create a mutable reference to a variable using something like the following snippet:

```
let mut x = 1;
let y = &mut x;
*y = 1
```

As expected this updates the value of x. However, a mutable reference is not really usable in case of an array due to limitations in the parsing. In particular, the parser prohibts the use of parentheses on the left-hand-side of an assignment. The following snippet does not compile:

```
let mut a = [1, 2];
let mut p = &mut a;
(*p)[1] = 1;
```

8.3 Ordering of Trait Implementations

Note Version 1

Sway makes a best-effort analysis to detect the dependencies among the various nodes of the AST, so as to type-check them in the proper order. However, it is important to note that for some constructs of the language such as traits the ordering isn't properly inferred. Therefore scripts like the following will be rejected by the compiler:

```
script;

trait MyTrait {
    fn foo(self, other: Self) -> Self;
}
fn main() -> () {
    let a = 8u8;
    let b = a.foo(2u8);
}

impl MyTrait for u8 {
    fn foo(self, other: Self) -> Self {
        if self > other {
            self
        } else {
            other
        }
    }
}
```

Here, the compiler fails to re-order the trait implementation before the `main()`.

8.4 Typing Based on Unreachable Expressions

Note Version 1

Consider the following example:

```
let x: u64 = {
    if cond() {
      return;
    } else {
      return;
    }
```

```
    42
};
```

In this example, `x` is always assigned to a `never` value since both branches of the `if`-expression return. However, the type of `x` is determined by the last expression of code block which is unreachable. This means that the compiler doesn't take into account the reachabiity of nodes during type inference. Also note we cannot type `x` with the `never` type.