Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Verification of Advanced Properties for Real World Vyper Contracts

Master Thesis

Christian Bräm

October 17, 2020

Advisors: Marco Eilers, Prof. Dr. Peter Müller
Programming Methodology Group
Department of Computer Science, ETH Zürich

Abstract

Ethereum is a decentralized distributed blockchain network that supports smart contracts. Smart contracts are autonomous programs running on the blockchain. These smart contracts facilitate agreements without the need of a trusted third party. With such agreements, smart contracts often store large funds. For Ethereum, there are two main languages in which smart contracts can be written: Solidity and Vyper.

In Ethereum, once a smart contract is deployed it cannot be changed any more. Therefore, 2VYPER was developed, a verifer for smart contracts written in Vyper. 2VYPER gives users a set of specification constructs and can automatically verify if a contract adheres its specification.

In this thesis, we extend 2VYPER to support the verification of advanced properties for real world contracts. We were able to improve the performance and stability of 2VYPER and introduced new specification constructs. Additionally, constant functions can now be used in specification. We also enabled the support for loop invariants and inter contract invariants, invariants that refer to state of multiple contracts. Further, 2VYPER can now verify real world contracts with stable performance also in the presence of nonlinear integer arithmetic.

We extensively tested all new features of 2VYPER and could successfully show that 2VYPER is now able to verify such advanced properties for real world contracts.

Acknowledgements

First of all, I would like to thank my supervisor Marco Eilers for his support and constructive discussions throughout the course of the masters thesis. With his feedback, he guided me greatly towards this final version of the thesis.

Additionally, I would like to thank Professor Peter Müller who gave me the opportunity to pursue this thesis in his group.

Last but not least, I want to thank my girlfriend, friends and family for the countless helpful discussions and the many hours of reading previous versions of this thesis.

Contents

Contents **iii**

1 Introduction **1**

2 Background **3**
2.1 Ethereum 3
2.2 Vyper 4
2.2.1 Structure 4
2.2.2 Loops 8
2.3 2VYPER 9
2.3.1 Verification 9
2.3.2 Back End 12

3 New Features for 2VYPER **15**
3.1 Performance: Avoid In-Lining 15
3.1.1 Motivation 15
3.1.2 Design 16
3.1.3 Encoding 20
3.1.4 Summary 21
3.2 Performance: Loop invariants 22
3.2.1 Motivation 22
3.2.2 Background 22
3.2.3 Design 23
3.2.4 Encoding 24
3.2.5 Summary 26
3.3 Pure Functions 27
3.3.1 Motivation 27
3.3.2 Design 27
3.3.3 Summary 34
3.4 Nonlinear Integer Arithmetic 35

3.4.1 Motivation . . . 35
3.4.2 Design . . . 35
3.4.3 Encoding . . . 38
3.4.4 Summary . . . 43
3.5 Inter Contract Invariants . . . 44
3.5.1 Motivation . . . 44
3.5.2 Background . . . 46
3.5.3 Design . . . 49
3.5.4 Summary . . . 59

4 Implementation **61**
4.1 Performance: Avoid In-Lining . . . 62
4.2 Performance: Loop invariants . . . 62
4.3 Pure Functions . . . 63
4.4 Nonlinear Integer Arithmetic . . . 64
4.5 Inter Contract Invariants . . . 65

5 Evaluation **67**
5.1 Performance: Avoid In-Lining . . . 67
5.1.1 Discussion . . . 69
5.2 Performance: Loop invariants . . . 70
5.2.1 Discussion . . . 72
5.3 Pure Functions . . . 74
5.3.1 Expressiveness . . . 74
5.3.2 Performance . . . 75
5.3.3 Discussion . . . 76
5.4 Nonlinear Integer Arithmetic . . . 77
5.4.1 Performance and Stability . . . 77
5.4.2 Annotation Overhead . . . 83
5.4.3 Discussion . . . 84
5.5 Inter Contract Invariants . . . 85
5.5.1 Discussion . . . 88

6 Conclusion and Future Work **89**

A Encoding of Loops in Pure Functions **91**

B Wrapped Integer Domain **93**

C Inter Contract Increase **103**

D Contracts Used for Evaluation **105**
D.1 Uniswap default method . . . 105
D.2 Modifying Struct in a Loop . . . 106
D.3 Altered Version of Stableswap's *get_y* . . . 107

D.4 Contract with Cubic Polynomial That Does Not Use Interpreted Lemmas 108

Bibliography **111**

Chapter 1

Introduction

Smart contracts are autonomous programs running on a decentralized distributed blockchain network. They facilitate agreements between multiple parties without the need of a trusted third party. These programs define some rules on how an agreement can be established. If the the conditions of a contract are fulfilled, the consequences are automatically executed.

The most prominent blockchain which provides such an open global computing platform is Ethereum [32]. There are multiple high-level languages for writing smart contracts for the Ethereum blockchain. The main two languages are Solidity [6] and Vyper [4]. All of the high-level languages for the Ethereum blockchain compile down to byte code which then gets executed in the Ethereum Virtual Machine (EVM).

The Ethereum blockchain, like most blockchains, implements the underlying concept of immutability. This means that it does not allow changes of the blockchain's past. Because of this immutability, Ethereum's implementation of smart contracts does not allow to alter a smart contract, once a smart contract is deployed. This also has the disadvantage that unintended behaviour, from the perspective of the programmer, cannot be fixed any more for a smart contract running already on a blockchain. The most well-known attack on a smart contract is the attack on *TheDAO* contract which resulted in the theft of $50M worth of Ether [18]. This attack was only possible because of a bug in the smart contract.

Therefore, there is great interest from writers of smart contracts to check their contracts. One way to check for bugs is testing. With tests, the presence of a bug can be shown only if there is a failed test. If all tests succeed there is no guarantee about all possible executions. Tests can be written with well-known unit test frameworks like pytest [9] or tools, which are specialised for smart contracts like Brownie [1], hive [3], vertigo [14] or Ethereum Tester [7].

Besides testing, a smart contract can be checked by searching vulnerable patterns in its code. Examples of tools which will report possible usages of vulnerable patterns are MythX [12], Slither [11], Oyente [8], Securify [10], SmartCheck [13] and ContractGuard [2].

Searching for vulnerable patterns does not give any formal guarantees. In addition, such direct searches of vulnerable patterns are incomplete because vulnerable patterns do not necessary imply the existence of a bug. If a contract designer wants stronger guarantees after checking his contract or wants to use some custom specification, there are now various verifiers for this. For example Act [5], solc-verify [21] and VerX [27] are verifiers for Solidity smart contracts. KEVM [23] is a verifier for the EVM byte code and can therefore be used for both Vyper and Solidity contracts but it requires learning a DSL to specify properties.

2VYPER [28] is a verifier for Vyper contracts. It enables verifying custom properties using special comments directly in the contract. The specification constructs were designed to give guarantees about the execution of a contract and the absence of security problems like missing access control or denial-of-service.

In this thesis we want to extend 2VYPER to enable the verification of more complex properties of real world Vyper contracts. We extended 2VYPER with the following features:

- We introduce loop invariants and private function specification to make the verification of 2VYPER more scalable
- We allow users to refer to constant functions of the smart contract in the specification
- We enable specifying and verifying inter-contract invariants
- We improve stability of verifying properties with nonlinear arithmetic

The thesis is structured as follows: Chapter 2 provides the general background for this thesis. In Chapter 3 each of the points from above are discussed focussing on their individual backgrounds, their problem statements and the verification and encoding of the problems. The implementation of these extensions to 2VYPER is the topic for Chapter 4. Subsequently, in Chapter 5 we evaluate all the new features of 2VYPER, with a conclusion in Chapter 6.

Chapter 2

Background

2.1 Ethereum

Ethereum is a distributed decentralised global computing platform which enables smart contracts. The yellow paper of Ethereum give a full formal specification of this platform [32].

Ethereum uses blockchain technology to store the state of its accounts and the past transactions. Miners provide new blocks and collect new transactions. The collected transactions are then grouped into the new mined block and are executed by the miners.

To compensate the execution cost, each operation in a transaction consumes a predefined amount of so called *gas*. The miner gets the used gas of a transaction times the gas price of the block as the reward.

Contracts and users can have accounts on Ethereum. Each account has a corresponding address. For user accounts, only their current Ether balance is stored. Contract accounts additionally contain their contract EVM byte code and some associated memory, which they may use to store arbitrary values.

A *transaction* can simply send some Ether from one user account to another. But it may also invoke some smart contract which makes some calculations or even calls other contracts and then possibly returns a value.

Calls to other contracts may also lead to a call back to the original contract. This is also called a reentrant call or *reentrancy*.

A transaction might also *revert* due to revert op-codes. Higher-level languages like Solidity and Vyper use these op-codes to e.g. provide the contract designer with an assert statement. If an assertion fails, the transaction reverts.

Beside the accounts and the transactions, there are also so called *events* that are stored in the blockchain. An event can be seen as logging of data in the blockchain that can be indexed and searched for. A user can therefore just look up the past event logs in the blockchain for free and does not have to query a contract and pay a gas price.

2.2 Vyper

Vyper is a nominally and statically-typed programming language for Ethereum smart contracts. The syntax is inspired by Python. For this thesis, we used Vyper in the version 0.1.0b17. In this section, we will discuss some language features. For more information about Vyper, please consider reading the documentation [17].

2.2.1 Structure

A Vyper smart contract consists of external contract declarations, event declarations, state variables, constants and functions.

State Variables

State variables store values that are persisted between transactions. They must be declared outside of any function. State variables are accessed via the `self` variable. A state variable is not allowed to define a default value.

Constants

Constants are also declared outside of any function. The constants must be assigned to their constant value at declaration site. They are replaced at compile time with their constant values at every usage location.

```
1 stored_data: int128
2 constant_data: constant(int128) = 42
3
4 @public
5 def __init__():
6   self.stored_data = constant_data
```

Listing 2.1: A simple contract with a state variable and a constant value.

In Listing 2.1, a simple contract is shown which uses a state variable and a constant value. `__init__` is a special function that initializes a contract on deployment.

Functions

In the functions, the state changes and calculations can be defined. Besides the already mentioned `__init__` function, `__default__` is another special function. `__default__` is the *fallback function* that is called when Ether is sent to a contract without specifying a specific function and when a non-existent function is called on a contract.

With the exception of the special functions, a function can declare if it can be called only by other functions from within the contract or only by external callers. The first type of functions are *private* functions, the latter are the *public* functions. The two special functions must always be public.

If a function does not modify any state variable, it can be declared *constant*. A *payable* function is open to accept some Ether. When a non-payable function receives Ether, the transaction reverts.

The `__default__` function must be payable and non-constant. The `__init__` function may be constant and payable.

```
1  stored_data: int128
2  constant_data: constant(int128) = 42
3
4  @public
5  def __init__():
6    self.stored_data = constant_data
7
8  @public
9  @constant
10 def get_data() -> int128:
11   return self.stored_data
12
13 @private
14 def increase_by_value(val: int128):
15   self.stored_data += val
16
17 @public
18 def increase_by_one():
19   self.increase_by_value(1)
20
21 @public
22 @payable
23 def __default__():
24   raise "Please do not send me Ether."
```

Listing 2.2: A simple contract with a state variable, a getter and two increase functions.

The example of Listing 2.2 is an extended version of the previous example. There is now a getter function, `get_data`, to retrieve the current value of the `stored_data`. Since this `get_data` function does not modify any state, it is declared to be constant. There is also an `increase_by_one` function which calls a private function `increase_by_value`. The `__default__` function of Listing 2.2 would accept some Ether because of the payable decorator, but it will always revert a transaction when called.

In Vyper, direct and indirect recursion is forbidden for private functions. A function may only call a private function if the private function was declared before the function itself.

Vyper functions can also be declared to be *non-reentrant*. Non-reentrant functions can specify a lock that they want to use. If a non-reentrant function is called, the function automatically tries first to acquire the specified lock. If it fails, the transaction reverts. This prevents reentrancy for these non-reentrant functions.

External Contract Declaration

If a contract wants to call another contract, there are two possibilities in Vyper. Either it makes a `raw_call` or calls it via a function declaration of an interface. With `raw_call`, an arbitrary function can be called from any contract. The drawback for this can be seen on line 9 to 13 of Listing 2.3. The result size must be specified and the conversion to an integer must be done manually. Additionally, Vyper would check the function name as well as the argument and return types of a function, if there would be a declaration. With a `raw_call` nothing gets automatically checked by Vyper.

```
1  contract FooBar:
2    def calculate() -> int128: constant
3    def change_stuff(): modifying
4
5  @public
6  def test(some_address: address):
7    v1: int128 = FooBar(some_address).calculate()
8
9    raw_v2: bytes[32] = raw_call(
10     some_address,
11     method_id('calculate()->int128', bytes[4]),
12     outsize=32)
13   v2: int128 = extract32(raw_v2, 0, type=int128)
14
15   assert v1 == v2
```

Listing 2.3: A simple contract with an in-line external interface.

If there was a declaration, one can call these functions just like normal functions of the contract itself. There are two ways to get to another contract's function declarations.

The first one is with a so called *in-line external interface*.

In Listing 2.3, a contract `FooBar` is declared as an in-line external interface. It has two functions. `calculate` does not change the state of any contract and returns a signed integer value. `change_stuff` can modify the state of `FooBar` and may also call other modifying / non-constant functions to change states of arbitrary many contracts.

The `test` function of this contract calls `calculate` once via the declared function of `FooBar` and once via a `raw_call`.

Another way to get to the function declarations is by using an *interface*.

```
1  # FooBar.vy
2
3  @public
4  @constant
5  def calculate() -> int128:
6    pass
7
8  @public
9  def change_stuff():
10   pass
```

Listing 2.4: An example of a Vyper interface.

```
1  from . import FooBar
2
3  @public
4  def test(some_address: address):
5      FooBar(some_address).calculate()
```

Listing 2.5: A simple contract calling an external function.

The contract of Listing 2.5 uses the interface of Listing 2.4 to call `calculate`.

Events

There are two ways to *log an event* in Vyper, either via `raw_log` or by using a call on the `log` variable. Again like `raw_call`, a contract designer can log an arbitrary value but must also ensure on their own that the logged byte-data is in the correct format, so that it can be searched for.

For the call on the `log` variable, one has to first *declare an event*. Events must be declared before any state variable or function declaration of a contract.

Once declared, it can be used as often as wanted in all functions of the contract.

```
1 Payment: event({amount: uint256, sender: address})
2
3 total_paid: uint256
4
5 @public
6 @payable
7 def pay():
8   received_ether: uint256 = as_unitless_number(msg.value)
9   self.total_paid += received_ether
10  log.Payment(received_ether, msg.sender)
11
12 @public
13 def log_event(bb: bytes32, bt: bytes32, data: bytes[256]):
14  raw_log([bb, bt], data)
```

Listing 2.6: A simple contract with events.

In Listing 2.6, an event `Payment` is declared. It is used whenever `pay` is called.

In this example, the code also refers to the **`msg`** variable. This message variable stores the caller's address in the `sender` attribute and the sent Ether in the `value` attribute. Like the **`log`** and **`self`** variables, the **`msg`** variable is implicitly provided and can be used before ever declaring it.

2.2.2 Loops

A feature of Vyper is that for every function an upper limit of the used gas can be calculated. To enable this, it does not allow arbitrary loops. Vyper only allows *for-loops* over lists with a statically known size.

```
1 @public
2 def foo(x: uint256):
3   for i in range(x):
4     pass
```

Listing 2.7: An invalid Vyper contract with a for-loop over a list with variable size.

Listing 2.7 shows an invalid Vyper contract. The list returned by **`range(x)`** would have an arbitrary size. Only a range with a constant value as argument is therefore allowed in Vyper.

2.3 2VYPER

As already mentioned in the introductory chapter, in this thesis we extend 2VYPER [28] and add new features.

2.3.1 Verification

2VYPER already supports multiple verification constructs for Vyper contracts:

- Postconditions
- General postconditions
- Transitive postconditions
- Checks
- General checks
- (History) Invariants

In Listing 2.8, a function can be seen that returns its input plus one.

The first postcondition on line 1 captures this. The second postcondition is more advanced. In contrast to the first postcondition, which just repeats the function's body, it captures the fact that if there would be an overflow, a transaction would get reverted.

This example shows the special comments 2VYPER uses for specifications. Like in Python, comments begin with a number sign (#). The special comments continue with an at-sign (@).

Besides the verification constructs, like the postcondition (`ensures`), 2VYPER also introduced some special verification functions like `result`, `revert` and `success`. The first can be used to address the value returned by a function. The latter two can be used to access the flag that states whether the transaction has reverted or not.

```
1 #@ ensures: result() == x + 1
2 #@ ensures: x == MAX_INT128 ==> revert()
3 @public
4 def add_one(x: int128) -> int128:
5   return x + 1
```

Listing 2.8: An example of 2VYPER specification.

Postconditions, General Postconditions and Transitive Postconditions

A postcondition contains an expression which must be true at the end of a Vyper function, even if the execution of the function reverts. It can access the

state of the contract before the function was called via the `old(...)` function. We call the state that can be accessed via `old(...)` also *old state*.

A normal *postcondition* can be attached to a public function and it has to hold at the end of this function (e.g. line 9 of Listing 2.9). A *general postcondition* is a postcondition which has to hold at the end of every public function (e.g. line 7 of Listing 2.9). The *transitive postconditions* are general postconditions that are reflexive and transitive with respect to the old state (e.g. line 5 of Listing 2.9). These postconditions can capture how the state changes if an arbitrary number of function calls are made. This arbitrary number can also be zero if the a function reverts. That is why transitive postconditions also have to be reflexive.

```
1  i: int128
2  owner: address
3
4  #@ preserves:
5    #@ always ensures: self.i >= old(self.i)
6
7  #@ always ensures: self.i == old(self.i) + 1
8
9  #@ ensures: result() == self.i
10 @public
11 def pay_and_set_i() -> int128:
12   temp: int128 = self.i + 1
13   self.i = 0
14   send(self.owner, self.balance)
15   self.i = temp
16   return temp
```

Listing 2.9: A Vyper contract with examples of all three possible postconditions.

In this section, it was mentioned that the postcondition can only be attached to and have to hold for public functions. This is the case, since 2VYPER fully in-lined all private function calls. Therefore, only public functions could have specifications.

Checks and General Checks

During execution of a Vyper function, there are multiple points in the execution of the function that are *public states*. We define a public state to be a state that is visible to other contracts. The contract's state of a contract A can be potentially read by another contract and is thereby visible if the contract that is currently executing is not A.

At the beginning and the end of a public function, there is a public state. A public state is also created at every point the contract sends Ether to or calls a function from another contract.

Like postconditions, checks can also access an old state. For checks the old state refers to the previous public state. At the start of the function, the current state and the old state would be identical.

Checks need to hold whenever a public state is created with respect to the previous public state. A normal *check* can be attached to a public function. It has to hold in every public state of the function and any public state of called private functions with respect to the last public state. For example, in Listing 2.10, the check on line 11 also needs to hold at the point where the public state of the called private function `pay` is created by `send`.

General checks, like general postconditions, are checks that have to hold for every public function. An example of such a check is on lines 4-5 of Listing 2.10.

```
1  i: int128
2  owner: address
3
4  #@ always check: msg.sender != self.owner ==>
5    #@ self.i == old(self.i)
6
7  @private
8  def pay(amount: wei_value, to: address)
9    send(to, amount)
10
11 #@ check: msg.sender != self.owner ==> revert()
12 @public
13 def pay_and_set_i(amount: wei_value, to: address,
14                   new_val: int128):
15     assert msg.sender == self.owner
16     self.i = new_val
17     self.pay(amount, to)
```

Listing 2.10: A Vyper contract with examples of both possible types of checks.

Invariants

2VYPER supports normal invariants (e.g. line 4 of Listing 2.11) and also history invariants (e.g. line 5 of Listing 2.11). For the history invariants, one can use the `old(...)` function to access the previous state.

```
1 i: int128
2 owner: address
3
4 #@ invariant: self.i == 0
5 #@ invariant: self.i == old(self.i)
6
7 @public
8 def foo():
9     self.i = 42
10    # Do something without creating a public state
11    self.i = 0
```

Listing 2.11: A Vyper contract with an invariant example.

For any public states s_1 and s_2, where s_2 happens latter than s_1 or is s_1, all invariants have to hold with s_1 as the old state and s_2 as the current state. This means that invariants have to be reflexive and transitive.

2.3.2 Back End

2VYPER uses the Viper framework [16] as its back end. Viper is a language and tool-set that provides an architecture for the development of verification tools and prototypes.

It provides two back ends, one based on symbolic execution, *Silicon*, and one based on verification condition generation, *Carbon*. Both back ends, Silicon and Carbon, use *Z3* [19] as their theorem prover.

The Viper language is a simple sequential, object-based, imperative programming language that is designed to be an intermediate language. While having some primitive types like `Int`, it has support of adding new types via *domains*. A domain can contain a set of function declarations and a set of axioms. These function declarations, which are called *domain functions*, consist just of their signatures. They are constrained using the *axioms* of the domain.

In Listing 2.12, the definition of a pair for integers can be seen. It has two functions, one to create new pairs of integers, and one to retrieve the first element of such a pair.

```
domain Pair {
  function create(a: Int, b: Int): Pair
  function getFirst(p: Pair): Int
  // other functions

  axiom getFirst_ax {
    forall a: Int, b: Int :: {getFirst(create(a, b))}
      getFirst(create(a, b)) == a
  }
  // other axioms
}
```

Listing 2.12: A simple Viper domain for pairs of integers.

The axiom `getFirst_ax` contains an universal quantifier. It describes the fact that if one would call `getFirst` on a newly created pair, the first element would be returned. The expression in the curly braces is the *trigger* for this quantifier. So, only if this trigger `getFirst(create(a, b))` can be matched with an expression during the proof search, the knowledge `getFirst(create(a, b)) == a` can be gained for the expression. For more information about triggers, please consider reading [26].

Viper also supports *functions* outside of domains. These functions may have a *body*, *pre-* and *postconditions*.

```
function subtractOne(x: Int): Int
  requires x > 0
  ensures  result >= 0
{ x - 1 }
```

Listing 2.13: A simple Viper function that subtracts one.

The `subtractOne` Viper function is shown in Listing 2.13. This function has a body, pre- and postcondition. It guarantees that its result is non-negative by restricting the input to be larger than zero. Viper verifies that the postconditions can be proven to hold for the result of the function. If a function has no body, the result of a function is unconstrained and this proof is trivially true.

A Viper function can only have one expression in its body. To specify a callable with multiple statements, *methods* exist in Viper. In Listing 2.14, an example of a Viper method can be seen.

```
1 method foo(a: Bool) returns (r: Int)
2 requires true
3 ensures r == 0
4 {
5   if (a) {
6     r := 2
7     assume false
8   } else {
9     r := 0
10  }
11  assert !a
12 }
```

Listing 2.14: A simple Viper method which always returns zero.

Methods can also return a value and have pre- and postconditions. In the example of Listing 2.14, we also see `assume` and `assert` statements. With an `assume` statement we can assume that some expression is true. This means that the verification continues only for the traces for which the expression of the assumption is true. After assuming `false`, which is never satisfiable, the current branch is not checked any further. By disabling the first branch of the `if` on line 5, the assertion `!a` can be verified as well as the postcondition of the method `foo`.

Chapter 3

New Features for 2VYPER

The goal of this thesis is to make 2VYPER usable for complex real world contracts. For this we introduced new features for 2VYPER. In the following sections of this chapter, these new features are introduced.

3.1 Performance: Avoid In-Lining

To make the verification more scalable, the following two new features were added to 2VYPER:

- We enabled specification on private functions, so that they do not have to get in-lined any more
- We introduced loop invariants into 2VYPER to avoid unrolling large loops (see Section 3.2)

3.1.1 Motivation

The design of 2VYPER aims to provide contract designer an easy tool to verify their properties. If they for example only have an invariant they want to check, this invariant should be the only specification one needs to write down.

With this idea, there are only two options of handling private functions without specification. Either all the functions could get fully in-lined or we make an over-approximation and havoc all state information on a private function. A more fine-grained approach would need more input from the contract designer. There is no other third option, since 2VYPER is a sound verifier.

In Vyper, there is no recursion. If the precision of the model is more relevant than the performance, in-lining is a good strategy not to lose information and generate few false negatives. But if a private function is called multiple

times, possibly via different other private functions, the in-lined code takes a large portion of the generated Viper code.

As the decision concerning the trade-off between precision of the model and performance depends on the use case, we want to enable that contract designers can specify their private functions and get precise information and good performance.

The goal is therefore to enable postconditions on private function. Instead of in-lining the postconditions of the private function can then just be assumed.

3.1.2 Design

In the following subsections, the problems and design choices are outlined to realize this goal.

States

2VYPER always models two states, the current state of the blockchain / the contract and the previous public state.

Public functions always have a public state at the beginning and at the end of their functions. Therefore, the current state and the last public state would be the same at these points. Invariants need to hold at every public state, which is why it is possible to assume all invariants at the start of a public function.

A private function, on the other hand, does not have a public state at the start of its call. The last public state is possibly not the same as the current state.

Without the ability to assume any information at the start of the function, some other source of information is needed to constrain the current state.

For this we introduced *precondtions*. We allow preconditions only on private functions, since for private functions it is possible to check those precondition at every call site. For public function it would not be possible, since anybody could call the function with any arguments.

Listing 3.1 shows an example that demonstrates why we need some information about the current state. On line 8, the private function `send_some_ether` creates a public state by sending some ether. Therefore, all invariants must be shown to hold at this point. Without the knowledge of the precondition, the invariant `self.f == 3` could not be asserted.

```
1 f: int128
2
3 #@ invariant: self.f == 3
4
5 #@ requires: self.f == 2
6 @private
7 def send_some_ether(a: address):
8   self.f += 1
9     send(a, 123)
10
11 @public
12 def foo():
13   self.f -= 1
14   self.send_some_ether(msg.sender)
15
16 # [Some more functions, like a constructor that establishes
     the invariant]
```

Listing 3.1: A simple contract with an invariant and a private function with a public state.

This example of Listing 3.1 had a normal invariant. If a history invariant would have been used instead, it would not be possible to write a precondition with just the existing verification functions of 2VYPER and be able to verify the history invariant.

For history invariants or other specifications that refer to a previous public state, some knowledge about the current state in relation to the last public state must be provided as well.

The same issue also occurs at the end of a private function call. Since it is unknown if a public state was created inside of a private function, an over-approximation has to made when a private function gets called.

Therefore, we introduced a new specification function `public_old(...)`. An expression in `public_old` will always get evaluated in the context of the last public state.

Using this, we can show an example with history invariants. In Listing 3.2 pre- and postconditions for the private function `send_some_ether` can be seen. With the precondition, the history invariant on line 3 can be verified for the public state created due to the send on line 10. Using the postcondition on line 7, the history invariant can also be verified in the public state at the end of the public function `foo`.

```
1  f: int128
2
3  #@ invariant: self.f >= old(self.f)
4
5  #@ requires: self.f >= public_old(self.f)
6  #@ ensures: self.f >= old(self.f)
7  #@ ensures: self.f >= public_old(self.f)
8  @private
9  def send_some_ether(a: address):
10     send(a, 123)
11
12 @public
13 def foo():
14   self.f += 1
15   self.send_some_ether(msg.sender)
```

Listing 3.2: A simple contract with a history invariant and a private function with a public state.

Checks

Previously, checks could only be placed on public functions. But because of in-lining, all the called private functions had to fulfil the check as well. This means that a private function may has to satisfy multiple checks from different sources.

If private functions do not get in-lined and their specifications are just assumed, it is unclear whether or not a check of a calling public function is actually satisfied in the possible public states of the private function.

There are two solutions for this problem without changing the definition of a *check*. One would be to collect all the call sites of a private function, generate a conjunction of their checks and use this conjunction as the check of the private function. Another approach would be to allow specifying checks for private functions, where the checks of the private functions must imply the checks of the calling public functions.

To give the contract designer more expressiveness, we decided to realize the second approach.

Events

2VYPER checks event logging from one public state to the next following public state. This is one use case of checks in 2VYPER. For example in Listing 3.3, two events get logged. The check on line 5 says that if the transaction is successful (does not revert), one `Transfer(msg.sender, a, 42)` event gets logged before every public state.

```
1  Transfer: event({_from: indexed(address),
2                   _to: indexed(address),
3                   _value: uint256})
4
5  #@ check: success() ==> event(Transfer(msg.sender, a, 42), 1)
6  @public
7  def transfer_send(a: address):
8      log.Transfer(msg.sender, a, 42)
9      send(a, 123)
10     log.Transfer(msg.sender, a, 42)
```

Listing 3.3: A function which logs two events.

Every private function, even private constant functions, can log events. Additionally, every private function can have a public state. Notably, a private constant function can simply call a public constant function of an external contract to generate a public state.

Therefore, with our event handling, the information on how many events are currently logged needs to be passed into and out of the private function as pre- and postconditions, respectively.

```
1  Transfer: event({_from: indexed(address),
2                   _to: indexed(address),
3                   _value: uint256})
4
5  #@ requires: event(Transfer(sender, a, 42), 1)
6  #@ ensures: event(Transfer(sender, a, 42), 0)
7  #@ check: success() ==> event(Transfer(msg.sender, a, 42), 1)
8  @private
9  def send_some_ether(sender: address, a: address):
10     send(a, 123)
11
12 #@ check: success() ==> event(Transfer(msg.sender, a, 42), 1)
13 @public
14 def transfer_send(a: address):
15     log.Transfer(msg.sender, a, 42)
16     self.send_some_ether(msg.sender, a)
17     log.Transfer(msg.sender, a, 42)
```

Listing 3.4: A public function which logs two events and calls a private function that creates a public state.

An example for this can be seen in Listing 3.4. To fulfil the check in the public state created by the send on line 10, the information that there is currently one logged event is needed. This is provided by the precondition

on line 5. Further, this private function guarantees with its postcondition on line 6 that no such event got logged since the last public state.

3.1.3 Encoding

In this section, we show how private function calls are encoded from the caller's and the callee's perspective.

Caller

If the specification of a private function gets assumed, a caller needs to perform the following steps.

1. Assert the preconditions of the private function.
2. Havoc all the present knowledge about events.
3. If the function is modifying / non-constant, havoc the current as well as the last public state.
4. Ensure that the checks of the called private function are stronger than the checks of this function.
5. Assume the postconditions of the private function.

Callee

A private function needs to do the following steps differently than public functions. The full encoding of public functions can be seen on page 35 - 39 in [28].

- At the start of the private function, invariants are assumed for the last public state but not for the current state.
- The knowledge that there are currently no logged events at the start of the function is havocked as described in the next section.
- The preconditions get assumed.
- Invariants and checks at the end of the function do not get checked.

Havocking Events

To support assuming events and havocking all known knowledge about events, we reworked the event handling and event translation to Viper.

In the original event handling, all events are translated to Viper predicates. The argument of an event can be translated directly as arguments of the predicate.

If an event got logged in a Vyper statement, this statement got translated as inhaling one full permission to this predicate. In checks, it could be verified that a certain amount of permission to a predicate is available. If a public state is created, all permission to any event predicate is again exhaled, to freshly start the counting. The full original event handling is described in [28].

The original method takes advantage of the fact that at the start of a Viper method no permission to any predicate is provided. This models that there are no logged events at the start of public functions. This is no longer true for private functions. Therefore, this implicit knowledge has to get havocked.

For this havocking, we introduced the following method. A non-negative amount of the predicate that represents an event is inhaled with with unconstrained variables for every argument of the event. This models that this event may have been logged an unknown amount of time with unknown arguments.

If there is a pre- or postcondition that introduces some knowledge about an event, it is assumed that the unconstrained variables for the arguments of the event are not equal to the ones provided by the precondition.

For example, the precondition on line 5 of Listing 3.4 introduces the knowledge for the private function that the event `Transfer(sender, a, 42)` already got logged exactly once. When this precondition gets assumed, it must also be assumed that the previously mentioned unconstrained variables cannot be `(sender, a, 42)`.

Our event havocking strategy fails, if there is a disjunction of events while asserting them. For example the postcondition `ensures: event(Foo(1), 0) or event(Foo(2), 0)` could be verified even though the events were havocked with our method. Without inhaling anything, we have no permission to any predicate. Since we only used one set of unconstrained variables, the SMT solver can verify that the unconstrained variable cannot be both one and two and therefore `event(Foo(1), 0)` or `event(Foo(2), 0)` has to be true.

To successfully havoc knowledge about events, we are now disallowing the event-specification function at certain places e.g. in disjunctions.

3.1.4 Summary

We enabled specifications for private functions. Using these specifications, a private function call no longer has to get in-lined.

3.2 Performance: Loop invariants

3.2.1 Motivation

To be able to support loops without any input from the contract designer, there are two possibilities. To remain sound, either there is an over-approximation of the values of all local variables and parts of the state that might get modified inside a loop, for example by havocking them, or the loop gets unrolled and each iteration gets encoded separately in Viper.

Since Vyper only supports for-loops over lists with statically known size, all loops can be unrolled in a contract. As it is important not to lose any information about the state and generate possible false negatives, 2VYPER does unroll all loops.

This introduces at best a performance impact in the order of the number of statements in the loop times the number of loop iterations. At worst, it is exponential in the number of loop iterations.

To prevent loop unrolling without just havocking the state, some user input in the form of loop invariants is required.

The goal of this feature is to enable contract designer to use loop invariants, so that loops do not have to get unrolled.

3.2.2 Background

Loop invariants are in general assertions that have to hold at the start and at the end of each iteration. In Hoare logic [24] they are defined as follows.

$$\frac{\{C \wedge I\}\ \text{body}\ \{I\}}{\{I\}\ \textbf{while}\ (C)\ \text{body}\ \{\neg C \wedge I\}}$$

Assuming the loop condition C and the loop invariant I, after an arbitrary iteration of the loop, the loop invariant I must still be true. If the loop invariant I holds before the loop and I holds after each loop iteration, we know after the loop that C is false and I is true.

As already mentioned, Vyper only supports for-loops over fixed sized lists. Therefore, the loop condition C would be that the current index in this list is smaller than the length of the list. But with break statements it is non-trivial to define the $\neg C$. After the loop, the index in this list might still be smaller than the length of the list.

3.2.3 Design

We want to enable contract designers to use the loop variable, current loop iteration, the list we iterate over and the previously iterated elements in the loop invariants.

Previously Iterated Elements

In for-loops, the iteration is always over the elements of the list. Consequently, these elements are important. For example in Listing 3.5, to be able to write an invariant that precisely describes the loop's behaviour one must be able to refer to the previously iterated elements.

```
1 #@ ensures: res == 42
2 @public
3 def foo():
4     res: int128 = 0
5     for i in [1, 3, 37, -3]:
6         #@ invariant: res == old(res) + sum(previous(i)) +
      loop_iteration(i)
7         res += i + 1
```

Listing 3.5: A simple loop which adds all numbers in the provided list.

To solve this, we introduced the new specification funciton `previous(...)`. This example in Listing 3.5 shows a loop invariant that tries to capture the exact current value of `res`. For this, the `previous(i)` elements get accessed to sum them up.

The `old(...)` function was also used in this example. We defined the old state to be the state before the loop. We extended the definition of the `old` function to also support the access of non state variables. So, in the context of loop invariants, the `old(...)` function returns the value of a variable it had before the loop. In the previous example it would be 0 for `old(res)`.

The last new specification function is the `loop_iteration(...)` function. We introduced this function to be able to access the current loop iteration.

Both `loop_iteration` and `previous` take the loop variable as argument. We designed these functions in this way so that it is possible to access this information also in the case of nested loops.

Loop variable

Since Vyper only supports for-loops, we wanted to simplify the loop invariants for contract designers. Loop variables have no value after the last iteration of a for-loop. The index would be out-of-bound.

```
@public
def foo(a: int128):
    res: int128 = 0
    for i in range(300):
        #@ invariant: loop_iteration(i) < 300 ==> res == i
        res += 1
```

Listing 3.6: A simple loop which adds one to a variable in each iteration.

One way to solve this problem would be to require users to always wrap usages of the loop variable into an implication, as show in Listing 3.6.

As already mentioned in the background, it is hard to define a loop condition for loops with break statements. To solve this problem as well as the problem with the loop variable, we introduced the following encoding for loop invariants.

3.2.4 Encoding

```
// <For each of the local variables used in the loop, declare
    a new version>
// Other variable declarations

// Code before the loop
// Base case: Known property about loop variable
assume <loop index> == 0
<loop variable> := <iterated list>[<loop index>]
// Check loop invariants before first iteration of the loop
assert <loop invariant 1>
assert <loop invariant 2>
assert <loop invariant ...>

// Havoc current and old state
self := <some fresh variable>
contracts := <some fresh variable>
old_self := <some fresh variable>
old_contracts :=<some fresh variable>
<loop index> := <some fresh variable>
<loop variable> :=<some fresh variable>
// From this point on, the new version for each of the local
    variables in the loop is used.
// <Assume invariants and type assumptions for both the
    current and the old state>
// <Assume type assumptions for each of the new versions of
    the local variables>
```

```
23 // Step case: Known property about loop variable
24 assume <loop index> >= 0 && <loop index> < <length of the
       iterated list>
25 <loop variable> := <iterated list>[<loop index>]
26 // Assume loop invariants
27 assume <loop invariant 1>
28 assume <loop invariant 2>
29 assume <loop invariant ...>
30
31 // Loop body
32
33 // End of loop body
34 label continue
35 <loop index> := <loop index> + 1
36 if (<loop index> == <length of the iterated list>) {
37     goto break
38 }
39 <loop variable> := <iterated list>[<loop index>]
40 // Check loop invariants for iteration <loop index> + 1
41 assert <loop invariant 1>
42 assert <loop invariant 2>
43 assert <loop invariant ...>
44
45 // Do not model this further
46 assume false
47
48 // After loop
49 label break
```

Listing 3.7: Our encoding of loop invariants.

With this encoding, the negated loop condition does not have to be assumed after the loop directly. A break is just translated as a `goto` to the `break` label. Also, if the last iteration of the loop was reached, there is a jump to the `break` label. Therefore, the negated loop condition can be seen as all possible conditions on how to leave a loop and, with our design, this knowledge is implicitly available at the `break` label.

With this design, the loop invariants are only checked at the end of a loop iteration, if it stayed in the loop. This means that loop invariants are not checked after the last loop iteration. This solves the problem of referring to the loop variable. Therefore, with this design, the loop variable can always be used in a loop invariant also without a guarding implication.

3.2.5 Summary

We introduced loop invariants in 2VYPER. With this, loops do not have to get unrolled any more.

Our custom design of the loop invariants enables loop invariants also in the presence of non-trivial control flow, introduced by `break` statements, and gives contract designer the ability to refer to previous iterated elements using `previous(...)`.

3.3 Pure Functions

3.3.1 Motivation

Frequently, some important properties of a contract are an aggregation of the contract state or a calculation depending on an input. These properties can be captured in a Vyper contract using private, constant functions.

To be able to reason about these properties without manually encoding them into specifications, we want to provide an alternative. Private functions without any side-effects should be possible to use in specifications.

Therefore, we want to enable that a contract designer can refer to private, constant functions without side-effects in specifications.

3.3.2 Design

Private, constant functions of a Vyper contract are not necessarily side-effect free. As already mentioned, they may still log events. There are also private, constant functions that do not even return a value. There are only two purposes for such functions.

Either they just log events or they are in some sense a guard and revert a transaction if some properties are not fulfilled. Without a return and the ability to alter the state, there is no other use case for such a function.

Private, constant functions are normal functions. They may use any Vyper statement as well as read out state variables. Loops and conditions are no exception.

Side-Effects

A call to a function in the specification must not change anything. Private, constant functions fulfil this criterion, with the exception that they may log an event. Therefore, we introduced a new decorator `@pure`. If a function has this decorator, 2VYPER checks that it is private, constant and does not log any events. In Listing 3.8, an example of such a decorator can be seen.

```
1 #@pure
2 @private
3 @constant
4 def add_1(i: int128) -> int128:
5   return i + 1
```

Listing 3.8: An example usage of a pure decorator.

Well-Definedness

Like all functions in Vyper, private, constant functions might revert. If the function reverts, the result of a function is not defined.

To solve this problem, we extended the definition of the following specification functions:

- `success`
- `revert`
- `result`

The original definitions of these functions were that the `success()` and `revert()` returned a bool that represented if the current function was successful or reverted. The `result()` function returned the result value of the current function.

In our new versions of these functions, they still behave the same when no argument is provided. But these specification functions can now also take a pure function call as an argument.

The `success(...)` and `revert(...)` functions return true or false if the pure function call in the argument succeeds or reverts, respectively. The `result(...)` function returns the result of the pure function call.

```
1  #@pure
2  @private
3  @constant
4  def add_1(i: int128) -> int128:
5    return i + 1
6
7  #@ ensures: success(self.add_1(5))
8  #@ ensures: result(self.add_1(41)) == 42
9  #@ ensures: revert(self.add_1(MAX_INT128))
10 @public
11 def foo():
12   pass
```

Listing 3.9: A Vyper contract with pure functions in postconditions.

The contract of Listing 3.9 shows how these functions take the function call as an argument. All three postconditions can be verified. If there is an over- or underflow in any step of a calculation, the transaction reverts. This is why the call in the last postcondition `self.add_1(MAX_INT128)` would revert.

Encoding

One use case for Viper functions is that they can be used to encode pure functions, so that they can be used in specifications.

Therefore, all three of the altered specification functions, `success`, `revert` and `result`, are encoded as Viper functions. The `result(...)` function is encoded additionally with a precondition that the function call did not revert. If the `result(...)` function is used with a pure function call as an argument that can revert, this mentioned precondition will be violated and 2VYPER reports this as a verification failure.

The definition of these specification functions requires that the result value and a success flag are returned at the same time.

As it is not possible to return two values at once in Viper, we use a struct as the return value, in which both the success flag and the real result are saved.

In simplified terms, a struct is just a data structure that can be accessed with a Viper function. This Viper function returns a value given some index and a struct object. For example, if `o` is a struct object, `struct_get(o, 4)` will return the value at slot four of the struct. The full struct encoding is described in [28].

The pure functions themselves are also encoded as Viper functions. However, Viper functions only support one expression as their body and not arbitrary statements with a possible non-trivial control flow. This is a problem since the bodies of pure functions may contain arbitrary statements including for-loops. For-loops, especially with loop invariants, cannot get translated into just one expression. Therefore, we use Viper functions without a body to encode the pure function and use the postconditions of the Viper function to constrain the resulting struct.

In the Vyper code, the function's statements determine its result. Therefore, we must translate these statements into assertions that we can use to constrain the result of the corresponding Viper function in its postcondition.

For example, in Listing 3.9 the one statement in the `add_1` function constrains the result to be `i + 1`. This will be translated to a postcondition in the encoded Viper function. Such a postcondition can be seen on line 6 of Listing 3.10.

```
1  function p$add_1(self: Struct, i: Int): Struct
2    // [Some type assumptions about the self struct and the
       argument i]
3    // Overflow checks
4    ensures (struct_get(result, 2): Int) == i + 1 >= MIN_INT128
       && i + 1 <= MAX_INT128
5    // The statement as a postcondition
6    ensures (struct_get(result, 3): Int) == i + 1
7    // The success flag
8    ensures (struct_get(result, 0): Bool) == !(struct_get(
       result, 2): Int) ? true : false
9    // The returned result value
10   ensures (struct_get(result, 1): Int) == (struct_get(result,
       3): Int)
```

Listing 3.10: An encoding of the *add_1* function.

The signature of the encoded Viper function is very similar to its counterpart in Vyper with the exception that it takes also the `self` variable as an argument, since it may read the current state, and returns a struct as previously mentioned. The slots zero and one of this returned struct are always reserved for the success flag and the result value, respectively.

There are four main types of Vyper statements: assignments, assertions, conditionals and loops. The encoding of each of them is explained in the following list.

- **Assertions**: Assertions check that a condition is true or else revert the transaction. `raise` statements behaves like `assert false`.

  If some assertion fails, the next statements should not get executed. Since we encode all statements as postconditions and all of them are assumed at the same time, we have to use a way to disable some postcondition modelling the statements, which should no longer get executed. This can be seen on lines 6 and 9 of Listing 3.12. These two encoded statements will no longer constrain the result struct if the condition of the assertion, stored in slot 4, is false.

- **Assignments**: Since all postconditions describe a single state, a variable used in a postcondition cannot have two different values.

  For example if we encode the following Vyper code: `a = 1; a = 2`, we cannot translate it into `ensures a == 1` and `ensures a == 2`. This cannot be satisfied. We have to use a single assignment form, for example `ensures a$1 == 1` and `ensures a$2 == 2`.

  In postconditions of Viper functions, only the arguments and the spe-

```
#@pure
@private
@constant
def foo():
  # [Some previous statements]
  assert i == 0
  i += 1
  assert i == 1
  # [Some later statements]
```

Listing 3.11: Parts of a pure Vyper function with an assignment.

```
function p$foo(self: Struct): Struct
  // [Some previous postconditions]
  ensures (struct_get(result, 4): Bool) ==
    !((struct_get(result, 3): Int) == 0)
  // [Some checks for overflow]
  ensures !(struct_get(result, 4): Bool) ==>
    ((struct_get(result, 5): Int) ==
      (struct_get(result, 3): Int) + 1)
  ensures !(struct_get(result, 4): Bool) ==>
    (struct_get(result, 6): Bool) ==
      !((struct_get(result, 5): Int) == 1)
  // [Some later postconditions]
```

Listing 3.12: Parts of an encoded pure Vyper function with an assignment.

cial `result` variable can be used. No new variables can be created. But structs can have arbitrary many slots. Therefore, we use different slots of result struct as quasi local variables.

The variable `i` of Listing 3.11 has, during the first assertion, the slot with index 3 in the encoded counterpart Listing 3.12. After the assignment, the variable gets a new slot with index 5.

There are multiple assignment statements. All are encoded similarly. `return` statements are encoded also as an assignment to a result variable, with the addition that all statements that would get executed after the return, get encoded as a postcondition that is trivially true and does not constrain the result struct.

- **Conditionals**: If-statements in Vyper can be translated as a ternary expression for each changed variable. But an arbitrary statement can possibly not be simplified using ternary expressions.

  Again, the postconditions must be true for a single state. Therefore, if

we encode a conditional we have to disable some postconditions that model the branch that is not taken.

All the statements in the if-scope therefore get the following implication. *Under the assumption that the if-condition is true*, constrain the result struct according to the given statement. An example for this can be seen in Listing 3.14. The if-condition is stored in slot 5 and the variable `i` was stored in slot 3 before the condition. On lines 7 to 8 of Listing 3.14, the fact that `i` is incremented is encoded. But this only needs to hold if the if-condition is true. For this, the implication on line 6 is used. An else-case would work similarly.

At the end of the if-statement, the mentioned ternary expression merges the modified variables back together (e.g. lines 9 - 12 of Listing 3.14).

```
1 #@pure
2 @private
3 @constant
4 def foo(a: bool):
5   # [Some previous statements]
6   if a:
7     i += 1
8   # [Some later statements]
```

Listing 3.13: Parts of a pure Vyper function with a condition.

```
1  function p$foo(self: Struct): Struct
2    // [Some previous postconditions]
3    ensures (struct_get(result, 5): Bool) ==
4      (struct_get(result, 2): Bool)
5    // [Some checks for overflow]
6    ensures (struct_get(result, 5): Bool) ==>
7      ((struct_get(result, 6): Int) ==
8        (struct_get(result, 3): Int) + 1)
9    ensures (struct_get(result, 9): Bool) ==
10     ((struct_get(result, 5): Bool) ?
11       (struct_get(result, 6): Int) :
12       (struct_get(result, 3): Int))
13   // [Some later postconditions]
```

Listing 3.14: Parts of an encoded pure Vyper function with a condition.

- **Loops**: Loops without loop invariants can be modelled as nested conditionals. With a loop invariant, we perform similar steps as mentioned in Section 3.2, but without ever asserting the loop invariants.

  Viper functions without a body cannot be used to assert anything, just to assume something.

To keep 2VYPER sound, if the Vyper function contains a loop invariant or something else to verify, we decided that a Vyper method must be generated as well. This Viper method is solely responsible to check that e.g. the loop invariant actually can be established and is preserved. The encoding of the Vyper function to a Viper method is described in [28]. It is then safe to assume the loop invariants in the Viper function.

`continue` and `break` statements introduce non-trivial control flow. They are encoded as follows.

For `continue`s, the condition under which the `continue` statement is reached, is stored and all statements after the `continue` statement are encoded as postconditions that are trivially true. Return statements are handled similarly.

After a loop iteration, all modified variables get merged together using ternary expressions like in the encoding for if-statements, just using the condition under which the `continue` was reached. In addition, an expression is generated, which is an aggregation of the `continue` conditions using exclusive-ors. This expression is also assumed at the end of a loop iteration and models the fact that for each loop iteration only one `continue` statement can be reached.

The `break` statements are encoded similarly for the whole loop and not just a loop iteration.

An example of a loop encoding with a break statement can be seen in Appendix A. The pure Vyper function can be seen in Listing A.1. If `True` is passed as an argument, this function `foo` returns zero. Otherwise, it will return 42.

Some postconditions in the encoded Viper function in Listing A.2 are omitted. Therefore, to give some context, the following slots contain already some data. In slot 9 of the result struct, the `res` variable is stored before the loop. In slot 8, the loop variable is stored. The index of the loop iteration is stored in slot 6. The argument `a` was stored in slot 2.

On lines 4 - 6 and 8 - 9 the encoded loop invariants can be seen. Every assertion of translated Vyper statements in the loop only need to hold under the condition that the loop invariants are true. This can be seen with the implication e.g. on lines 12 - 13.

Only one break out of the loop can happen. This fact gets encoded using the exclusive or-ed conditions under which a break out of the loop can be reached. This can be seen on lines 30 - 41.

The merge of all modified variables can be seen on lines 44 - 56.

Besides using the pure function in the specification, these private, constant functions are also used in regular Vyper code. To encode a call to a pure function, we do not need to in-line the pure function even though pure functions are not allowed to have specifications. We can simply use the encoded form of the pure function, a Viper function, and encode the call to the pure function just as a call to this Viper function.

3.3.3 Summary

We have shown an encoding of pure Vyper functions that supports all statements that can occur in such function, including for example assertions, assignments, conditionals and loops

Using this, pure functions can be used in every specification: preconditions, postconditions, transitive postconditions, invariants, history invariants, loop invariants and checks.

3.4 Nonlinear Integer Arithmetic

3.4.1 Motivation

As already mentioned in Chapter 2, Viper uses Z3, an SMT solver, as its final back end.

Vyper does not support any floating point type. Its decimal type is also just a scaled integer type. Therefore, all number types of Vyper are integers.

Hilbert's tenth problem is the challenge to come up with an algorithm that can decide, for any polynomial equation with integer coefficients and finite number of unknowns, whether there exists a solution for the equation with all unknowns taking integer values. This problem was shown unsolvable in 1970 by Yuri Matiyasevich. [25]

Therefore, nonlinear integer arithmetic is undecidable for an SMT solver. However, Z3 still manages sometimes to come up with proof that a property holds. For this it uses some heuristics. For example, if all variables are bound, with an upper and lower bound, and an encoding of the problem in bit-vectors is possible, Z3 would pass these vectors to a SAT solver to get to a solution.

These heuristics of SMT solvers are very unstable. Small changes to a problem can mean that an SMT solver can no longer verify a property. Sometimes just a random factor in the SMT solver causes a verification to succeed or fail.

For a more stable support of nonlinear integer arithmetic, we provide a new way to handle expressions containing nonlinear integer arithmetic in 2VYPER.

3.4.2 Design

To solve this problem of undecidability, part of the problem must get relaxed. One approach is to map everything into the space of real numbers. For real numbers, nonlinearity is decidable. [30]
But the precision of the result would be lost. The mapping back to the integer space cannot be made exactly. Only a region of possible values could be returned. [31]
Another approach is to use uninterpreted functions instead of the interpreted multiplication, division and modulus operations. By doing this, an expression containing these uninterpreted functions seems to be linear for an SMT solver. To get a precise model, these uninterpreted functions must get constrained as much as possible.

The mapping of the problem to uninterpreted function does not solve the undecidability. User input is needed. The user input is in form of derivation

steps. The SMT solver can do simple reasoning steps automatically but needs guidance from the user on which steps to perform. To enable a user to give this needed information, we introduced lemmas for 2VYPER.

We decided to realize the latter approach. With these uninterpreted functions, we want to prevent Z3 from using its heuristics.

Our approach of using uninterpreted functions and lemmas is greatly inspired by IronClad [22].

Lemmas

We enabled two types of lemmas: in-line lemmas and lemma definitions.

- **In-line Lemmas**: These lemmas are statements containing a boolean expression. An example can be seen in Listing 3.15 on line 4.

```
1 #@ ensures: success() ==> result() == x * x + x
2 @public
3 def foo(x: int128) -> int128:
4   #@ lemma_assert x * (x + 1) == x * x + x * 1
5   return x * (x + 1)
```

Listing 3.15: Function using an in-line lemma.

- **Lemma Definitions**: These are reusable lemmas that might have multiple steps. In Listing 3.16, a lemma definition can be seen on lines 1 - 2.

  This lemma definition is used in an in-line lemma (line 7) to enable the verification of the wanted property.

```
1 #@ lemma_def dist(x: int128):
2     #@ x * (x + 1) == x * x + x * 1
3
4 #@ ensures:  success() ==> result() == x * x + x
5 @public
6 def foo(x: int128) -> int128:
7   #@ lemma_assert lemma.dist(x)
8     return x * (x + 1)
```

Listing 3.16: Function using a lemma definition.

We enabled the usage of lemma definitions in all specifications, not just in-line lemmas. Pure functions might also contain nonlinear arithmetic. If a pure function is used, some extra information might be needed. This extra information can be provided using lemma definitions.

An alternative version of the example from Listing 3.16 can be seen in Listing 3.17. The `foo` function is now a pure function. Without allowing lemmas in

the postcondition, `success(self.foo(x)) ==> result(self.foo(x)) == x * x + x` could not be verified.

```
#@ lemma_def dist(x: int128):
    #@ x * (x + 1) == x * x + x * 1

#@pure
@private
@constant
def foo(x: int128) -> int128:
    return x * (x + 1)

#@ ensures: lemma.dist(x) and success(self.foo(x)) ==> \
    #@ result(self.foo(x)) == x * x + x
@public
def bar(x: int128):
  # [Some function implementation]
```

Listing 3.17: Pure function needing a lemma definition.

Interpreted

There is a class of cases where Z3 actually performs well also in the presence of nonlinear integer arithmetic. And there is another class of cases where a lot of lemmas would be needed to verify a property using uninterpreted functions.

Since these two classes are largely overlapping according to our evaluation (see Section 5.4), we introduced an interpreted scope for both in-line lemmas and lemma definitions. Everything in an interpreted lemma scope gets asserted using the interpreted functions and then, to still gain the information for the uninterpreted functions, also assumed for the uninterpreted functions.

Contract designers can now decide on their own if they want to use the default uninterpreted scope or if they want to use an interpreted scope.

Listings 3.19 and 3.18 show an in-line lemma and a lemma definition respectively using an interpreted scope. For a lemma definition, all steps inside the lemma are considered to be interpreted, if the lemma definition has this `interpreted` decorator.

```
#@ ensures: success() ==> result() == x * x + x
@public
def foo(x: int128) -> int128:
  #@ lemma_assert interpreted(x * (x + 1) == x * x + x * 1)
  return x * (x + 1)
```

Listing 3.18: Function using an in-line lemma.

```
#@ interpreted
#@ lemma_def dist(x: int128):
    #@ x * (x + 1) == x * x + x * 1

#@ ensures: success() ==> result() == x * x + x
@public
def foo(x: int128) -> int128:
  #@ lemma_assert lemma.dist(x)
    return x * (x + 1)
```

Listing 3.19: Function using an interpreted lemma definition.

3.4.3 Encoding

In Viper, uninterpreted functions that are globally constrained are domain functions with their corresponding axioms. In those axioms, often quantifiers are used to express that something holds for all objects of that domain type. These quantifiers have triggers.

Wrapped-Integer

To define these *domain* functions, first, a new domain has to be created that models integer numbers. We will call this domain the *wrapped-integer domain* from now on and the corresponding type or an object of this type *wrapped-integer*.

There are two options how to construct this domain.

- Any operation that can be performed on an integer is also modelled in the wrapped-integer domain (e.g. addition, greater than comparison).
- We define a destructor to get a normal built-in integer given a wrapped-integer so that the existing operations from the built-in integers can be reused.

To not introduce unnecessary axioms and overhead, we decided to realize the latter.

Of course, a constructor for the wrapped-integer is also needed to convert an integer into a wrapped-integer.

```
1  domain wInt {
2    function wrap(x: Int): wInt
3    function unwrap(x: wInt): Int
4
5    axiom wrap_ax {
6      forall i: Int :: {wrap(i)} unwrap(wrap(i)) == i
7    }
8
9    axiom wrap_unwrap_ax {
10     forall i: wInt :: {wrap(unwrap(i))} wrap(unwrap(i)) == i
11   }
12
13   // other functions and axioms
14 }
```

Listing 3.20: The wrapped-integer domain with a constructor and destructor.

In Listing 3.20, the constructor and the destructor for wrapped-integers are shown. With these axioms, both `wrap` and `unwrap` get constrained.
`unwrap` is always needed when we use a wrapped-integer for an operation that requires an integer. `wrap` happens only when we really need an integer to be a wrapped-integer. Therefore, `wraps` are more rare than `unwrap`. That is why we trigger only on `wrap`.

With these functions, it is possible to switch between the wrapped-integers and normal integers.

The other functions that are left to be modelled are the nonlinear functions on integers: multiplication, division and modulo.

For any of these operations, their mathematical definition is captured using individual axioms. We also added some convenient axioms to provide the same level of expressiveness as [22].
These axioms express the single reasoning steps mentioned above.

The whole wrapped-integer domain with all the axioms is shown in Appendix B. We proved most of them correct. Some axioms relay on the fact that Vyper uses a symmetrical modulus. These axioms cannot be proved directly using Viper, since Viper uses the mathematical modulus.

Background: Matching Loop

For the performance it is crucial that no *matching loop* can happen for triggers. This means that there should be no trigger in an axiom that matches with multiple different expressions in an axiom.

In the following, we will use an example from [29]. Please note that we do not use this `Nat` domain of these examples in 2VYPER.

The domain in Listing 3.21 contains a matching loop. The `plus` function should model an addition, the `succ` and `pred` functions model the successor and predecessor of a natural number respectively and the `tag` function models whether a value is non-zero.

```
1  domain Nat {
2     function plus(m: Nat, n: Nat): Nat
3    // other functions
4
5    axiom plus_def {
6      forall m: Nat, n: Nat :: {plus(m, n)}
7        plus(m, n) == (tag(m) == 0 ? n : succ(plus(pred(m), n))
       )
8    }
9    // other axioms
10 }
```

Listing 3.21: An axiom with a matching loop.

First, an instantiation for `plus(m, n)` can lead to an instantiation of `plus(pred(m), n)`. This can again lead to an instantiation of `plus(pred(pred(m)), n)` and so on.

A solution for this is to allow only one instantiation at a time e.g. with *limited* functions.

```
1  domain Nat {
2     function plus(m: Nat, n: Nat): Nat
3     function plusL(m: Nat, n: Nat): Nat
4    // other functions
5
6    axiom plus_def {
7      forall m: Nat, n: Nat :: {plus(m, n)}
8        plus(m, n) == (tag(m) == 0 ? n : succ(plusL(pred(m), n)
       ))
9    }
10   axiom plus_limited {
11     forall m: Nat, n: Nat :: {plus(m, n)}
12       plus(m, n) == plusL(m, n)
13   }
14   // other axioms
15 }
```

Listing 3.22: A domain using limited functions to prevent matching loops.

In Listing 3.22, the solution using limited functions can be seen. The trigger of the `forall` in the `plus_def` axiom will no longer match `plusL(pred(m), n)` and with that there is no matching loop. The `plus_limited` axiom models the fact that the limited plus function is just the plus function with another name.

Also like in this example, limited functions have the same arguments and also the same return type as their normal-function-counterparts. In addition some axioms introduce the knowledge that these two functions are actually the same (here the `plus_limited` axiom). By having two functions modelling the same thing, we can trigger on the normal function and use the limited function in the axiom, where the normal function would lead to matching loop.

If arbitrarily many instantiations with matching loops are disabled using limited functions, the instantiations must be guided by hand.

For example, if one wants to assert that `plus(succ(succ(x)), x) == succ(succ(plus(x, x)))` holds, first `plus(succ(succ(x)), x) == succ(plus(succ(x), x))` must be asserted to trigger the needed instantiation. The user input that we require does exactly this. It helps the SMT solver with the needed instantiations.

This problem of having matching loops, corresponds to the undecidability of nonlinear integer arithmetic for SMT solvers.

Usage of Wrapped-Integers

If all multiplication, division and modulo operations were replaced using the uninterpreted functions, the SMT solver surely would not have to use its heuristics for nonlinear integer arithmetic any more. But some information would get lost as well. Not every multiplication is nonlinear. For example, three times five is linear since both values are constant. Only when a nonlinear operation is performed with two variables modelling some unknown values, we should use the uninterpreted functions of the wrapped-integer.

Therefore, besides using the wrapped-integer domain as domain for the uninterpreted functions, we decided to use the wrapped-integer type as a marker as well. All integer variables that do not have just one constant value get wrapped-integer as their type.

All integer local variables are normal integers at declaration site. Integer arguments and state variables on the other side are always wrapped-integers.

Every integer local variable that is being influenced by non-constant values should be a wrapped-integer as well. Therefore, an integer local variable might also change its type to a wrapped integer if one of the following expressions is assigned to it:

1. an expression containing a wrapped-integer
2. an expression containing a function call
3. an expression that accesses a data structure
4. any expression, if the assignment is located inside of a conditional
5. any expression, if the assignment is located inside of a loop

All of these points are over-approximations. Regarding the first point of the listing above, $0 * x$, where x is a wrapped-integer, is an expression containing a wrapped-integer. The result however is constant.
For the second point, if a function that does not read a state variable and gets just constant values as its argument is called, only a constant value could be returned. Since we have no knowledge that a called function actually does not read some state, we have to make an over-approximation and transform also possible constant values to wrapped-integers.

It is not possible to track wrapped-integers inside of data-structures. Therefore, it has to be assumed that any integer stored in a data-structure is a wrapped-integer. In our encoding, we always use wrapped-integers in this case.
Since Vyper has no knowledge about wrapped integers, e.g. in a list of integers, integer arguments as well as integer constants can be stored. On the Viper side, this would mean that it is a data-structure where wrapped-integers and normal integers can be stored. For this to work, it would need a common super type of wrapped-integers and normal integers as well as subtyping in general for Viper. This does not exist.
To continue with the example of over-approximations, for point three, it would be to access a stored constant integer. This would again result in a constant value that does not have to get converted to a wrapped-integer.

The second to last point models the fact that if a variable inside of an *if* gets modified, the variable has multiple possible values after the *if*. It is an over-approximation if the condition of the conditional would be a constant value. This is a rare case, since this *if* is either unnecessary or leads to dead code. If constant if-conditions would be used in real world contracts, this over-approximation could be easily detected and removed.

The last point is analogous to the previous. Again, if a loop iterates e.g. over a list that happens to have only constant values, no variable inside the loop could get assigned to an unknown value just from these constant values.

All of these points show how an integer variable should change its type to wrapped-integer in some cases. But a variable can only have one type in Viper. Since Viper variables are just a model for the actual Vyper variables, two Viper variables can also be used to model one Vyper variable. Therefore, in our encoding if the type of an integer variable is changed, a new variable

gets created with the wrapped-integer type. This new variable is then used from this point on.

If such a type change would happen inside of a conditional or a loop, the new variable gets created already before the conditional or loop and is assigned just with the current value of the original variable.

We can leave linear operations interpreted and therefore we only use the uninterpreted multiplication, division and modulus when both operands are wrapped-integers.

If at least one of the operands is not a wrapped-integer or another operation like addition or subtraction is used, the wrapped-integers are unwrapped.

We optimized our encoding, so that we do not generate `wrap(unwrap(...))` pairs. Without losing any precision or information, these pairs could be dropped.

There are only two cases where an integer has to get wrapped back to a wrapped-integer.

- If an integer expression gets stored in a variable. This can only happen using assignment or return statements.
- If an integer expression is used as an operand for a multiplication, division or modulo operation.

The first case is necessary to not lose the wrapped-integer marker type. The uninterpreted multiplication, division and modulo functions need wrapped-integers as argument. Therefore, the second case is needed.

3.4.4 Summary

We introduced uninterpreted functions that model multiplication, division and modulus. To relax the undecidable problem of solving nonlinear integer formulas with an SMT solver, we introduced lemmas as user inputs. To not create matching loops, limited versions of the uninterpreted functions are introduced as well. With limited functions, the lemmas are needed as they can provide the needed helping steps to guide the SMT in the right direction.

The interpreted version of the multiplication, division and modulus can still be used when the user explicitly requests this to offer the best of both worlds.

3.5 Inter Contract Invariants

3.5.1 Motivation

Invariants capture properties of a contract that have to hold in every public state of the contract. Previously, 2VYPER only supported invariants accessing information of one contract.
But many popular smart contracts like Uniswap [33] or Stableswap [20] also maintain invariants between multiple contracts.

These inter contract invariants capture properties of another contract also in relation to this contract. For example the two mentioned contracts, Uniswap and Stableswap, are smart contracts specialised for trading with other contracts. Therefore, inter contract invariants are vital to verify the correctness of the contracts.

An example for inter contract invariants can be seen using Listing 3.24 and 3.23. The `simpleIncrease` is a smart contract that stores an integer value for every possible caller using a map. This value can be accessed using the `get` function and modified using the `increase` function.

The `interContractIncrease` contract has two `simpleIncrease` tokens. In the `__init__` function of the `interContractIncrease` contract, the contract stores an initial difference of the two token values in `_diff`. This contract always modifies the value of both of its tokens in the same way. Therefore, an inter contract invariant could be that the amount of `token_A` minus the amount of `token_B` is always equal to the initial difference stored in `_diff`.

```
1  # simpleIncrease
2
3  amounts: map(address, uint256)
4
5  @public
6  def increase() -> bool:
7      self.amounts[msg.sender] += 1
8      return True
9
10 @public
11 @constant
12 def get() -> uint256:
13     return self.amounts[msg.sender]
```

Listing 3.23: A simple contract that stores an amount for each unique address.

```
# interContractIncrease

import simpleIncrease as Token

token_A: Token
token_B: Token
_diff: uint256

@public
def __init__(token_A: address, token_B: address):
    self.token_A = Token(token_A)
    self.token_B = Token(token_B)
    value_A: uint256 = self.token_A.get()
    value_B: uint256 = self.token_B.get()
    assert(value_A >= value_B)
    self._diff = value_A - value_B

@public
def increase() -> bool:
    result: bool = False
    if self.token_A.get() != MAX_UINT256 \
      and self.token_B.get() != MAX_UINT256:
        self.token_A.increase()
        self.token_B.increase()
        result = True
    return result
```

Listing 3.24: A contract that maintains a constant difference between two *simpleIncrease* contracts.

A challenge for inter contract invariants is that they have to hold even though any other contract can call any contracts without the knowledge of other contracts. For example, using the listings from the previous example, any contract can call the `simpleIncrease` contract without the knowledge of the `interContractIncrease`.

But in this example the mentioned inter contract invariant actually holds, since only the `interContractIncrease` can modify its value in the `amounts` map of the `simpleIncrease` contract.

This example contains an unrealistically large amount of information for the `interContractIncrease`. The whole implementation of the `simpleIncrease` contract is available. Usually, if a real world contract uses another real world contract, only the interface and possibly a documentation of the other contract is publicly available.

Therefore, we want to enable proving inter contract invariants, needing only information that is available for a contract designer.

Consequently, a further challenge for inter contract invariants is that we cannot require the total knowledge of all contracts. We can only require the full code of the contract that gets verified and interfaces for other contracts.

3.5.2 Background

Model of Other Contract State

2VYPER not only models the state of the current contract using a struct type, it also uses a map to model all other account states on the blockchain.

Slightly simplified, a map in Viper is a function that takes a key and returns the last stored value for that key. The map storing all account states would have the address of the account as key and a struct modelling the account state as value.

The full map encoding can be seen in [28].

Interfaces

Interfaces cannot declare any state variables, just their public functions. Because of this, only postconditions were allowed for interfaces in 2VYPER. But these postcondition might still need to express some information about the properties of the contract. To be able to still reference a property that might have been defined using pure functions or state variables, *ghost functions* were introduced to 2VYPER.

```
1  # simpleInterface
2
3  #@ interface
4
5  #@ ghost:
6      #@ def val() -> int128: ...
7
8  #@ ensures: success() ==> result() == val(self)
9  @public
10 def get() -> int128:
11   raise "Not implemented"
```

Listing 3.25: An interface with a ghost function.

The interface of Listing 3.25 shows a declaration and usage of a ghost function. The ghost function just models a property, here some value `val`, for all `simpleInterface` contracts. On line 8 of this listing, the ghost function is

used to ensure that this contract returns the value `val` of the contract itself. For this, it passes `self` as an argument to the ghost function.

Besides using an interface for its declaration, an interface can also be implemented. If there is any ghost function declared for an interface, the contract implementing it must also define these ghost functions. For example, in Listing 3.26, the state variable `value` is used as an implementation.

A state variable would be intuitive as an implementation for the ghost function, but it would also be a valid implementation to just use a constant like the one of Listing 3.27. The contract only needs to ensure that whatever gets returned in `get` is the same as the value of the ghost function.

```
1  from . import simpleInterface
2  implements: simpleInterface
3
4  value: int128
5
6  #@ ghost:
7      #@ @implements
8      #@ def val() -> int128: self.value
9
10 @public
11 def get() -> int128:
12   return self.value
```

Listing 3.26: An implementation of the simpleInterface using a state variable as an implementation for the ghost function.

```
1  from . import simpleInterface
2  implements: simpleInterface
3
4  value: int128
5
6  #@ ghost:
7      #@ @implements
8      #@ def val() -> int128: 42
9
10 @public
11 def get() -> int128:
12   self.value = 42
13   return self.value
```

Listing 3.27: An implementation of the simpleInterface using a constant value as an implementation for the ghost function.

The interface used the ghost function with `self` as the argument. But it is also possible to call ghost functions using another address. In Listing

3.28, a contract is shown that uses the ghost function `val` with `self.i` as the argument.

```
1 from . import simpleInterface
2
3 i: simpleInterface
4 v: int128
5
6 #@ ensures: success() ==> self.v == val(self.i)
7 @public
8 def __init__(a: address):
9   self.i = simpleInterface(a)
10  self.v = self.i.get()
```

Listing 3.28: An example of an inter contract postcondition.

The encoding of these ghost functions is a Viper function that takes a struct, modelling the wanted contract state, as well as other potential further arguments, and have the wanted property encoded as an expression in the body.

Therefore, the encoding of the postcondition from Listing 3.25 on line 8 would be `assert succ ==> res == g$val(self_address(), (self_address() == self_address() ? self : map_get(contracts, self_address())))`.

The ghost function is translated to a Viper function `g$val`. Its first argument is the `self_address()` because the ghost function is called with `self` as its argument. The second argument is the state of the contract at this address. 2VYPER uses a separate struct to model *self*. Because of this, there is a ternary expression. The variable `contracts` is the previously mentioned map that stores a struct for every address.

Invariants at the End of a Public Function

The postcondition on line 6 of Listing 3.28 is an inter contract postcondition. It refers to some state from another contract. This postcondition can be verified. But a corresponding invariant e.g. `self.v == 0 or self.v == val(self.i)` may not hold.

Postconditions need to hold at the end of a function. With the postcondition of `get` we know at the end of `__init__` that `self.v` is actually the same as `val(self.i)`.

Invariants on the other hand need to hold in all public states of the contract, also if the contract is currently not called.

A public state is a state in which the contract might be readable. At all points in time, where the contract is currently not executing code, it might

be readable. For example, after finishing the execution of a public function, a contract returns. This begins a public state until this contract is called again. In this time arbitrary many other contracts in the blockchain might have been called and changed their state.

Therefore, to assert an invariant at the end of a public function, nothing can be assumed from any other contract. The information about the state of other contracts is encoded as the `contracts` map in Viper. Therefore, this `contracts` map has to get havocked before asserting the invariants.

Invariants at External Calls

At the start and the end of a public function there is a public state, but also if an external function call is made. External function calls were handled in 2VYPER the following way. The full encoding and further description can be in Chapter 4.4 of [28].

1. Evaluate the arguments of the call.
2. Check and update the Ether balance if some Ether is sent.
3. Assert the checks of the function.
4. Assert the invariants of the contract without the information of the `contracts` map.
5. Handle reverts of the called function.
6. If the called function is not constant, havoc `self` and `contracts`.
7. Assume the invariants and the transitive postconditions for the havocked states.
8. If we have a postcondition from an interface, assume this postcondition.

Point five of this enumeration models the fact that at any point the called function might revert. This can happen due to the lack of gas in the transaction, an assertion violation, an under- or overflow, etc. If the called function does revert, the whole transaction reverts.

3.5.3 Design

Invariants for Interfaces

A first step is to enable users to specify invariants for interfaces. With those, some information can be gained from other contracts.

After havocking the `contracts` map, it would be possible to assume all invariants for all known interface references. With that, simple inter contract invariants would be possible.

For example, if there is an invariant in the `simpleInterface` of Listing 3.25 that states that the expression `val(self)` is always e.g. 42, it will be possible to add an invariant in Listing 3.28 like `self.v == 0 or self.v == val(self.i)`.

Known Interface References

We define an interface reference to be a variable that holds an address and has an interface type. For example the state variable `i` on line 3 of Listing 3.28 would be an interface reference.

The known interface references is the set of all interface references that are accessible at a point in time.

Caller Private Expression

With only invariants in interfaces, some inter contract invariants would still not be able to get verified. For example, it is still not possible to verify the inter contract invariant in the motivation section, Listing 3.24.

A reasonable invariant for the `simpleIncrease` interface would be that the amounts of all addresses are non-decreasing. But this does not help with the verification of the inter contract invariant of `interContractIncrease`. The problem is that we want to ensure that if we are not calling `simpleIncrease`, the amount of the `interContractIncrease` stays the same, i.e. that no other contract can change this amount.

An invariant has no notion of a caller. It is a property of a contract and not a function. An invariant has to hold even if the contract is currently not called. Therefore, such a property cannot be captured using an invariant.

It could expressed as a general check in the form of:

```
forall({a: address}, msg.sender != a ==>
    some_expression_using_a == old(some_expression_using_a))
```

This check asserts that in every public state only the caller can modify its expression and for all other addresses the expression stays the same. An example for `some_expression_using_a` would be `self.amounts[a]` for the `simpleIncrease` contract of Listing 3.23.

A caller, e.g. the `interContractIncrease` contract, can then assume, for all interface references that are currently not called, that `some_expression_using_a` stays the same. For example, the expression stayed the same for `token_B` when `increase` gets called on `token_A` on line 23, and analogously for `token_A` when `increase` gets called on `token_B` on line 24.

To simplify the notation, we introduced a new specification construct, *caller private*. Instead of writing such a check:

```
always check: forall({a: address}, msg.sender != a ==>
    some_expression_using_a == old(some_expression_using_a))
```

with caller private expressions it can be written as:

```
caller private: some_expression_using_a
```

Listing 3.29 shows an example caller private expression on line 6. The `caller` function is used as a replacement of the quantified variable `a`. Every caller private expression must at least once use the `caller` function.

```
1  #@ interface
2
3  #@ ghost:
4      #@ def mapping() -> map(address, uint256): ...
5
6  #@ caller private: mapping(self)[caller()]
7
8  #@ ensures: success() ==>  mapping(self)[msg.sender] \
9    #@ == old(mapping(self)[msg.sender]) + 1
10 @public
11 def increase() -> bool:
12   raise "Not implemented"
13
14 #@ ensures: success() ==> result() \
15   #@ == mapping(self)[msg.sender]
16 @public
17 @constant
18 def get() -> uint256:
19   raise "Not implemented"
```

Listing 3.29: The interface for the *simpleIncreaseContract*.

Conditional Caller Private Expression

With caller private expressions, we can model the state of a contract that can only be modified by the caller of a function. Each caller would have its own unique corresponding expression.

In Vyper contracts, there are often some privileged addresses, e.g. the owner of the contract, that are the only addresses that may alter some state.

In Listing 3.30, an interface is shown that has a function `stop`. Only the owner of this contract can call `stop` successfully and switch the `stopped` flag to true.

`stopped(self)` is a caller private expression, but only if the caller is the owner. This can be captured using a conditional caller private expression e.g. line 7 of the Listing 3.30.

```
1 #@ interface
2
3 #@ ghost:
4     #@ def owner() -> address: ...
5     #@ def stopped() -> bool: ...
6
7 #@ caller private: conditional(caller() == owner(self),
      stopped(self))
8
9 #@ ensures: msg.sender != owner(self) ==> revert()
10 #@ ensures: success() ==> stopped(self)
11 @public
12 def stop():
13   raise "Not implemented"
```

Listing 3.30: An interface of a contract with an owner.

Caller Private Expressions with Inter Contract Invariants

When a public state is created e.g. at the end of a public function, as already mentioned, invariants have to be asserted. While the current contract is not called, any other contract can make arbitrary state changes. However, with the invariants and caller private expressions on interfaces, these changes are not completely arbitrary.

It is known that, while a contract is not called and is currently also not calling any other contract, the caller private expressions of all known interface references have to stay the same. Also, we can assume that the invariants for each of the known interface references hold.

Therefore, after havocking the state to model the arbitrary changes, we can assume all invariants of all known interface references and now it can also be assumed that all caller private expressions stayed the same with `self` as the caller.

The gained knowledge allows to verify inter contract invariants.

As mentioned earlier, invariants are assumed whenever a new public state is created. For example, non-constant external calls might change states of arbitrarily many contracts. Therefore, as described in the background of this section, we havoc the current state. After the havocking, since this is a new public state, all the invariants are assumed. However, inter contract invariants must not be assumed at this point. The state of the called contract

might have changed and an inter contract invariant might not hold. Therefore, during external calls, the inter contract invariants must be asserted and not assumed.

An example of what might go wrong if we assumed inter contract invariants can be seen using Listing 3.31 and 3.32. The invariant on line 6 of Listing 3.31 states that its amount in the `simpleIncrease` contract with address `self.i` is always greater or equal to one, if the `ready` flag was set to true.

With just the knowledge provided with the `simpleIncrease` interface of Listing 3.29, Listing 3.32 would be a valid implementation. On line 15 of Listing 3.32, a public state is created and the amount of the current caller is set to zero.

This would break the invariant on line 6 of Listing 3.31. Therefore, this invariant must not be assumed during the external call of increase on line 18.

```
1  from . import simpleIncrease
2
3  i: simpleIncrease
4  ready: bool
5
6  #@ invariant: self.ready ==> mapping(self.i)[self] >= 1
7  #@ invariant: old(self.ready) ==> self.ready
8
9  @public
10 def __init__(a: address):
11   self.i = simpleIncrease(a)
12   self.i.increase()
13   self.ready = True
14
15 @public
16 def foo():
17   assert self.ready
18   self.i.increase()
```

Listing 3.31: A contract using an invariant with state of another contract.

```
1  from . import simpleIncrease
2  implements: simpleIncrease
3
4  amounts: map(address, uint256)
5
6  #@ ghost:
7    #@ @implements
8    #@ def mapping() -> map(address, uint256): self.amounts
9
10 @public
11 def increase() -> bool:
12   temp: uint256 = self.amounts[msg.sender]
13   self.amounts[msg.sender] = 0
14
15   # [Create public state]
16
17   self.amounts[msg.sender] = temp + 1
18   return True
19
20 @public
21 @constant
22 def get() -> uint256:
23   return self.amounts[msg.sender]
```

Listing 3.32: An implementation of *simpleIncrease* that sets the amount of the caller first to zero and creates a public state when called increase.

We have to split ordinary invariants and inter contract invariants.

The invariants are left the same as they were before. They cannot refer to other contacts' state. Inter contract invariants, on the other hand, can refer to other contracts' state. Before asserting them, the invariants and caller private expression are assumed for all known interface references. But the inter contract invariants themselves can only be assumed at the start of a function. For every public state created by external calls, these invariants must be asserted.

With this, the inter contract invariant of `interContractIncrease` can be verified. A full version of this contract can be found in Appendix C.

State Handling During External Calls

To check inter contract invariants, we create a specific encoding of several states that represent different points in the execution of the external call.

To simplify the descriptions, we introduce the following notation. The contract that makes the external call is called the *main contract*. We call the external call of the main contract the *initial external call*. The called contract is called the *receiver*. When *caller private expressions* are mentioned, we mean only the caller private expressions of the known interface references of the main contract with the main contract as the caller.

Reentrant calls are very important for this encoding. A receiver can of course change its state and therefore also its caller private expressions. But with reentrant calls also the state of the main contract and all other caller private expressions might get changed as well.

Reentrant calls also do not have to happen directly by the receiver. The receiver might call another contract and this contract might then make a reentrant call back to the main contract.

Therefore, in our encoding we split the modelled states into three parts:

- The first phase begins at the start of the initial external function call. It ends at the start of the public state of the external function call of the receiver that leads to a reentrant call back to the main contract.
- The second phase starts where the previous phase ended. This phase ends at the point where the last external function call of the receiver, which led to reentrant calls back to the main contract, returned.
- The third and final phase starts again where the previous phase ended and ends at the end of the initial external function call.

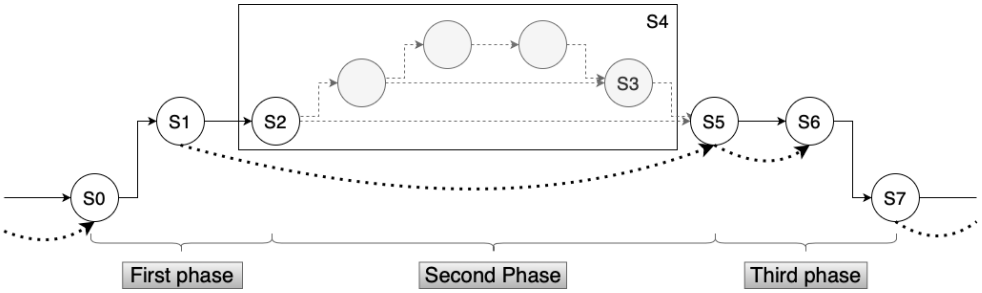
Please note that the second and third phase are optional. If no external call of the receiver leads to a reentrant call back to the main contract, there would be no such phases and the first phase would reach until the end of the initial external call. To still model all three phases is an over-approximation.

In each of these three phases, there might be public states. With our encoding, we model all possible public states. In our encoding we need to model seven states to be able to capture all possible public states.

The three phases and the seven states can be seen in Figure 3.1. The description of all of the seven states is in the following enumeration. Every item in the list corresponds to a new state.

0. Before the external call, there is a current state s_0. Each of our modelled states consist of a `self` struct (st_0), modelling the main contract state, and `contracts` map (c_0), modelling all the other accounts in the blockchain.
1. The state s_1 models the first public state of the initial external call in the first phase. The receiver might have made any local changes to his state and is now calling an external function.

Figure 3.1: A graphical representation of the three phases and the seven modelled states of the encoding of external calls. The solid arrows show where an execution has to go through. The dashed lines show possible execution paths. The big dotted arrows represent where in our encoding the inter contract invariants are asserted. The start of the arrow would be the old state and the head of the arrow the current state for which the inter contract invariants have to hold.



The receiver could have made arbitrary changes. But, since this state is in the first phase, we know that all the caller private expressions, except the ones of the receiver itself, stayed the same and the state of the main contract stayed the same ($st_1 == st_0$). Since this is a public state, all invariants of any contract have to hold in this state with s_0 as old state.

2. The state s_2 models any further public states, where the receiver performs its own external calls, in the first phase of the initial phase.

   This state is created analogously as the previous state, with the difference that now s_1 is the previous state instead of s_0.

3. The state s_3 models the point in time, where the last reentrant call returns. At this state, one or more reentrant calls could have happened.

   This state is not necessarily at the point where an external function returned to the receiver, but just at the point where the last reentrant call back to the main contract returned. s_3 is a state of the second phase.

   Since the main contract is called back with reentrant calls, we cannot assume that any caller private expressions or the main contract state itself stayed the same. The main contract must guarantee, how these values got modified, using its transitive postconditions and invariants.

   The invariants and transitive postconditions of the main contract are assumed for s_3 using s_2 as the old state.

4. s_4 is an artificial state. It combines the state where there were reentrant calls and the state where no reentrant call happened.

   An undefined boolean value decides for s_4 to be s_2 or s_3.

5. The state s_5 models the point where the last call, that may have caused a reentrant call, returned back to the receiver.

   s_5 is the start of the third phase. Since no more reentrant calls can happen, we can assume that the caller private expressions stayed the same as in s_4 and the main contract state stayed the same ($st_5 == st_4$).

   For this state we know that the caller private expressions of the receiver also stayed the same, since the initial external call did not execute any code since s_4. The invariants of all contracts can be assumed with s_4 as the old state.

6. s_6 models the public states of the third phase. Like in s_1 and s_2 the receiver might again modify its state and any other state of a contract that is not the main contract or is captured by the caller private expressions, except the caller private expressions of the receiver.

   Again, the invariants of all contracts can be assumed using s_5 as the old state.

7. The last modelled state s_7 is the point, where the external call returns back to the main contract.

   It is modelled analogously as s_6, just with s_6 as the previous state instead of s_5.

We over-approximate and model all possible public states of the receiver with the minimal information available for each of these states.

For example, it might be that a receiver does not create a public state before making a reentrant call. In this case the state s_1 would be identical with s_0. Our modelling of s_1 allows this, but it does not require it. With this state handling we only keep the information that is guaranteed by the specifications.

The big dotted arrows in Figure 3.1 show where we assert the inter contract invariants. The start of the arrow is the old state and the target of the arrow the current state.

With this design, it is known that inter contract invariants hold in s_5 with s_1 as the old state and s_6 with s_5 as the old state if the verifier can verify the specification.

To abbreviate the notation, we introduce (s_x, s_y). This means the s_x is the old state and s_y is the current state.

To show that the inter contract invariants hold in every possible public state of the initial external call, we have to show that with this design the inter contract invariant also hold in (s_0, s_1), (s_1, s_2), (s_2, s_5), (s_6, s_7). Since inter contract invariants, like all invariants, are transitive and reflexive, all remaining combinations of old and current states also hold.

Proof (Inter contract invariants hold for (s_0, s_1)) To prove this, we reduce this problem to showing that s_0 can be modelled with s_1 and s_1 can be modelled by s_5.

- s_1 is a copy of the state s_0 with some information havocked. Therefore, there exists for every unknown value in s_1 an exact value so that s_1 would be identical to s_0.
  s_1 can be therefore seen as an over-approximation of s_0 and thus can be used to model s_0.
- The same argument can be applied why s_2 models s_1. There exists a value for the undefined variable in state s_4 that allows s_4 to be identical to s_2. s_5 is again a copy of s_4 with just some information havocked.
  s_5 can be therefore seen as an over-approximation of s_1 and thus can be used to model s_1.

Since it is known that the inter contract invariants have to hold in (s_1, s_5) or else the verifier would report an assertion violation, this reduction of (s_0, s_1) to(s_1, s_5) concludes this proof. □

Proof (Inter contract invariants hold for (s_1, s_2)) To prove this, we reduce this problem to showing that s_2 can be modelled by s_5.

Analogue to the previous proof, there exists a value for the undefined variable in state s_4 so that $s_4 == s_2$ and s_5 can bee seen again as a copy of s_4 with just some information havocked.
s_5 can be therefore seen as an over-approximation of s_2 and thus can be used to model s_2.

Since it is known that the inter contract invariants have to hold in (s_1, s_5) or else the verifier would report an assertion violation, this reduction of (s_1, s_2) to(s_1, s_5) concludes this proof. □

Proof (Inter contract invariants hold for (s_2, s_5)) To prove this, we reduce this problem to showing that s_2 can be modelled by s_1.

s_1 and s_2 are created the same way. Both are copies of the previous state with the same information havocked. Besides the implicit equality constraints by the copy, only the invariants of the contracts constrain s_2. Therefore, every exact value of s_2 has to be a copy of s_1. For every possible value an unknown value in s_2 can have, the counterpart in s_1 can have the same value. This is the case, since invariants are reflexive.

Since it is known that the inter contract invariants have to hold in (s_1, s_5) or else the verifier would report an assertion violation, this reduction of (s_2, s_5) to(s_1, s_5) concludes this proof. □

Proof (Inter contract invariants hold for (s_6, s_7)) To prove this, we reduce this problem to showing that s_6 can be modelled with s_5 and s_7 can be modelled by s_6.

- The exact same argument as in the previous proof can be used to show that s_7 can be modelled with s_6.
- This argument from the previous proof has to be slightly extended to show that s_6 can be modelled with s_5. For every unknown value of s_6, the corresponding value in s_5 can have the same value, except for the caller private expression of the receiver.

  In s_5, it is known that all caller private expressions including the ones from the receiver are identical as in s_4. But for every unknown value in s_6 that models the caller private expressions of the receiver, the corresponding value in s_2 can have the same value.

  Therefore, for every possible value of every unknown in s_6, there exists a value for the corresponding model in s_5 so that these values are identical.

Since it is known that the inter contract invariants have to hold in (s_5, s_6) or else the verifier would report an assertion violation, this reduction of (s_6, s_7) to(s_5, s_6) concludes this proof. □

The postcondition of the called external function is assumed using s_7 as the present state and s_0 as the old state. To reduce the over-approximation, we assert the inter contract invariants with (s_5, s_6) after the assumption of these postconditions.

3.5.4 Summary

We have introduced inter contract invariants, a new specification construct. Unlike ordinary invariants, these can also refer to state from other contracts.

For the inter contract invariants, we allow abstraction of the other contracts and only require the interface of them.

To enable this, contract designers can now use invariants, inter contract invariants and caller private expressions, besides postconditions in interfaces.

Chapter 4

Implementation

The topic of this chapter is the implementation of the previously discussed designs. The main focus lies here on the parts of the implementation that are not already set by the design.

Limitation

We were able to implement the designs of all new features while supporting all of the previous supported language features.

During the writing of this thesis, a new Vyper version *0.2.5* was released. Currently, 2VYPER does only support Vyper in the version *0.1.0b17*. Additionally, some of the previously unsupported language features are still not supported, e.g. a tuple type as a return type of a function or some Vyper function like `extract32`.

Background

2VYPER is purely implemented in Python. Because it depends on Viper for the verification back end, some Viper resources are needed as well. These resources are the common Viper code that is needed for the verification.

2VYPER uses five phases, preprocessing and parsing, transformation, analysis, translation, and verification. In the *preprocessing and parsing* phase, the Viper contract and the specification are prepared and parsed into an abstract syntax tree (AST). During *transformation*, changes to the AST are made e.g. constants are replaced with the corresponding values. In the *analysis* phase, the Viper contract is type- and structure-checked. The Viper code is generated in the *translation* phase and afterwards used in the *verification* phase as input for the verification back end.

4.1 Performance: Avoid In-Lining

Preprocessing and Parsing, and Analysis

The preprocessing and parsing step as well as the analysis step have to get updated, to also support preconditions.

Translation

The function translation does now disambiguate if it gets a public or private function and translates it accordingly. Besides translating private functions, the function translator is also extended to support the modular approach of handling these private functions according to the design in Chapter 3.1.

4.2 Performance: Loop invariants

Preprocessing and Parsing

Since Vyper is very similar to Python, a parser designed for Python is used to build the AST. Because some Vyper code and our specifications are not in a perfect Python-like syntax, some parts e.g. event declaration and inline contract declaration, and have to get prepared for parsing. The full description of the preprocessing and parsing phase can be found in Chapter 5.1 of [28].

Verification constructs are in comments and would get ignored by the parser. Therefore, the preprocessor changes specification, like pre- or postconditions, to special assignments so that the parser does can build an AST without the need of special rules.

Loop invariants are no exception to this, but these assignments are now inside of a function body. Therefore, the scanning for those special assignments must now also happen inside of a function body. The collected loop invariants are then associated with an AST-node of their corresponding loop.

Transformation

In the transformation phase, constant values are gathered and replaced wherever they are used. Previously, the `range` expression was only supported with one constant literal or one constant variable, but Vyper allows using any constant expression as well as range expressions not starting from zero.

To support all possible range expressions, the constant transformer is updated.

Analysis

All variables that are modified inside of a loop must get havocked before assuming the loop invariants. For this, we collect all variable names that might got modified during analysis of a function. These used names are then stored in the node representing this function.

Translation

In the translation phase, the statement translator is updated to support our design of loop invariants. The Viper expression representing the list that is getting iterated over as well as the Viper variable storing the index of the loop iteration get stored in the translation context. To not overwrite this information in the presence of nested loops, a map is used. The name of the loop variable is used as the key of this map.
Functions like `loop_iteration` have one required argument, namely a loop variable. During translation, this argument is not translated but used as an uninterpreted string for the mentioned map in the translation context.

4.3 Pure Functions

Analysis

As stated in Chapter 3.3.2 a function has to be private, constant and does not log event to be pure. Therefore, all contracts that use the pure decorator on functions that do not fulfil these conditions have to be rejected.

Translation

2VYPER uses multiple translators e.g. one for functions, statements, expressions.

Every translator can produce Viper code with side effects. Mostly, this is due to the fact that many expressions, e.g. additions, can revert, which constitutes a side effect.

In pure functions, translation of expressions without side effects is needed. To not rewrite all translators, a mixin-class is introduced that updates the methods that produce Viper code containing side effects. Therefore, most of the needed translators just get a new pure version that inherits from the non-pure version and the mixin class.

The function translator and the statement translator had to be rewritten entirely.

4.4 Nonlinear Integer Arithmetic

To support stable nonlinear integer arithmetic, wrapped-integers are introduced. For this, the Viper file containing the wrapped-integer domain is added as a resource.

In Viper, wrapped-integers cannot directly be used in, e.g. an addition. First, the destructor has to get called to transform a wrapped-integer to an integer.

Therefore, existing 2VYPER code that works with integers needs to be updated to insert `wrap` and `unwrap` operations where needed. To not rewrite large portions of the translator, the functions that generate Viper AST-nodes were wrapped. This wrapper checks, if the node has some integer arguments, and unwraps all found wrapped-integers. This function stores in a flag, if some wrapped-integers were unwrapped, so that the result can be wrapped again if needed.

Preprocessing and Parsing, and Analysis

To support lemma definitions and in-line lemmas the preprocessing step as well as the analysis step has to get updated.

The in-line lemmas are changed to Vyper asserts in the preprocessing phase. They are then treated from this point on like Vyper assertions. The difference between lemmas and assertions is that integer expressions in lemmas are translated as if they were wrapped-integers and that they are always translated to Viper assertions.

Translation

For lemma definitions, a new translator is introduced.

The expression and arithmetic translator got updated to handle wrapped-integers. In the expression translator we implemented a way to avoid unnecessary wrappings of integers.
If the expression has e.g. multiple additions chained after each other, the results of the earlier additions do not have to get wrapped, only to get unwrapped again. For this, we introduced the notion of *top-level expressions*. These expressions are directly used in a statement. For example in the assignment `a = 1 + 2 + 3`, the top-level expressions would be `a` and `1 + 2 + 3`.

Inside of an expression, wrapped-integers are only unwrapped. If the top-level expression has an integer type and used wrapped-integers, only this expression is wrapped again.

The exception to this is that for multiplication, division and modulo, their operands are also treated as top-level expressions, since the uninterpreted

multiplication, division and modulus need wrapped integers as their arguments.

4.5 Inter Contract Invariants

Preprocessing and Parsing, and Analysis

To support inter contract invariants, caller private expressions and invariants on interfaces are needed.

The caller private expressions are treated similarly to general checks, with the encoding described in Section 3.5.

Invariants of interfaces are now also collected and are asserted when a contract implements the interface.

Translation

Inter contract invariants are mostly treated as normal invariants. Only in the expression and function translator these get disambiguated. The expression translator got updated to support the new encoding of an external call. The checking of the invariants at the of a public function got updated in the function translator.

Chapter 5

Evaluation

To evaluate each of these new features of 2VYPER, we tested the performance and usability of those features.

The measurements were performed on a Windows machine with 32GB DDR4 RAM and an i9-9900k CPU with a base frequency of 4.7 GHz and a boost frequency of 5.0 GHz.

Each configuration is run 50 times with the first ten runs as warm-up. In the plots, the mean and the 95% confidence interval of the remaining runs can be seen. For every run, the mean total time, calculated as the sum of all five phases of 2VYPER, was measured. Additionally, the mean time in the translation and verification phase is shown separately.

The first ten runs were omitted as a warm-up phase, because these first runs were usually slower due to the start up of the JVM for Viper.

As mentioned in previous chapters, Viper has two back end, Viper's symbolic execution back end (Silicon) and Viper's condition generation back end (Carbon).

If nothing else is stated, Silicon is used as back end.

5.1 Performance: Avoid In-Lining

This feature is designed for a faster verification of real world contract. Therefore, we assessed the time that is needed to verify a contract's specification with and without this new feature.

Previously, all private functions were fully in-lined. With the new feature, it is now possible to assume the postconditions of a private functions without the need to in-line it.

In order to assume a function's postconditions, the preconditions have to get asserted, some information about the current and previous public state has to get havocked and possible reverts have to get modelled. Also, there might be an encoding overhead when the private function is now also encoded as a Viper method.

If its faster to do all this steps than to verify the whole encoded function, it is expected that our new method is faster. If not, in-lining should be faster.

Set-Up

We performed the measurements of the in-lined and our new modular version for different numbers of calls on two contracts.

In Listing 5.1, the first contract is shown. The private function `inc_val` is called by the public function `f1`. This very simple private function just adds one to a state variable.

```
1  a: int128
2
3  #@ ensures: success() ==> self.a == old(self.a) + 1
4  @private
5  def inc_a():
6      self.a += 1
7
8
9  #@ ensures: success() ==> self.a == old(self.a) + 1
10 @public
11 def f1():
12     self.inc_a()
```

Listing 5.1: A simple contract that can increment its state variable.

The other contract can be seen in the Appendix D.1. This contract consists of copied parts from the Uniswap contract. The default function is the public function calling `ethToTokenInput`, a private function. The `ethToTokenInput` makes a further private function call to `getInputPrice`.

To not have multiple private calls that both might influence the measurements, `getInputPrice` is annotated with the pure decorator. As mentioned earlier, pure functions are not allowed to have specifications but get also not in-lined. A pure function call is encoded in Viper also with just a function call to their corresponding Viper function.

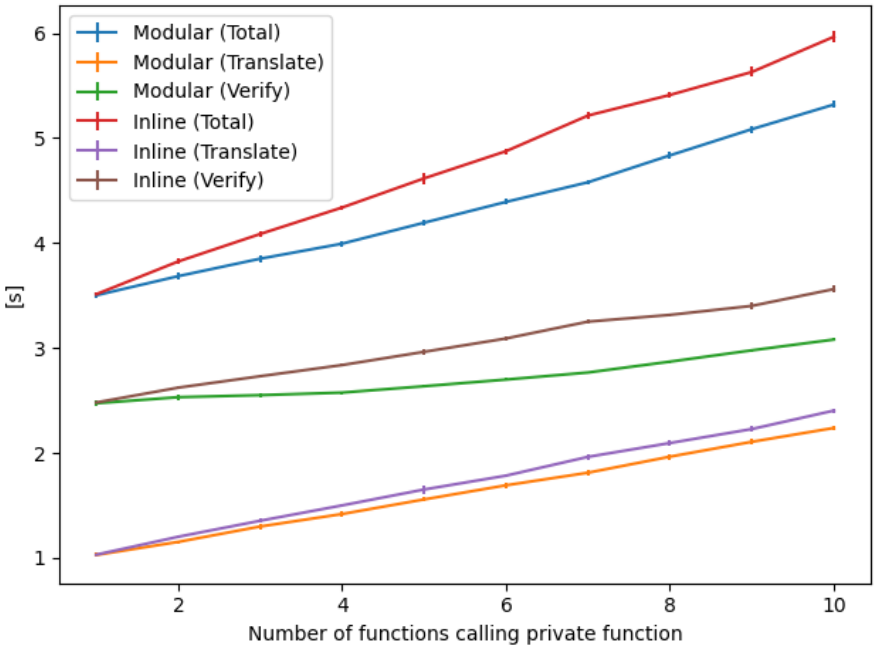
To simulate multiple calls to the private function, the public function in both contracts is duplicated.

To in-line the private functions, the postconditions of the private functions were removed.

Results

In Figure 5.1, it can be seen that our method outperforms in-lining. Both times seem to be linear, at least at this scale. Assuming the postconditions has a better scaling factor than in-lining. For every further public function calling the private function, our new method outperforms in-lining more.

Figure 5.1: Performance difference between in-lining and our new modular approach of handling of Uniswap's *ethToTokenInput* private function.



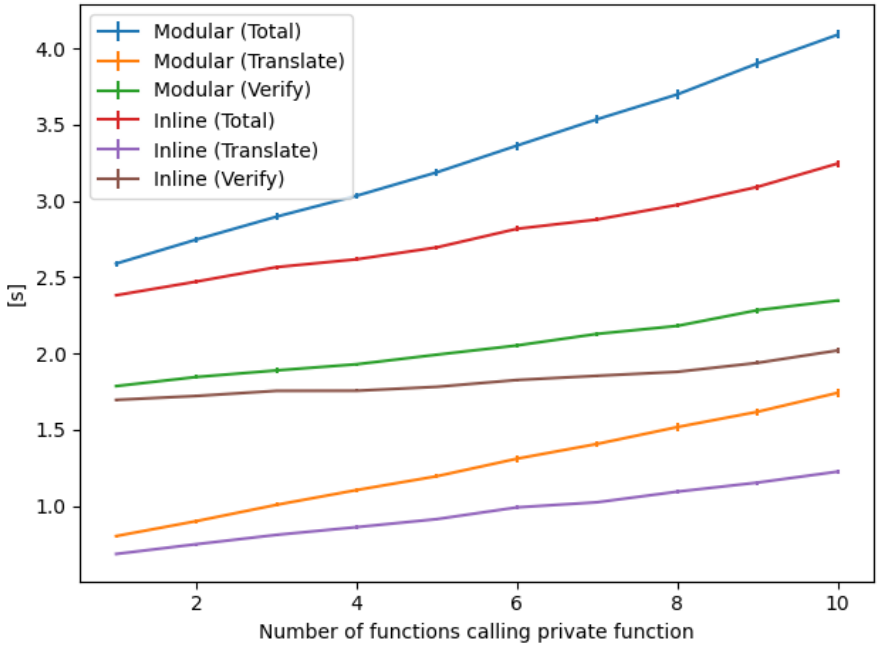
The limitation to our method can be seen in Figure 5.2.

5.1.1 Discussion

The results from our measurements show that if the function is non-trivial our function outperforms the previous in-lining strategy.

Besides the performance improvement for private functions, this new method enables specification for private functions, as this was not possible in the previous version of 2VYPER.

Figure 5.2: Performance difference between in-lining and our new modular approach of handling a simple private function.



5.2 Performance: Loop invariants

With loop invariants, verification of a loop should be possible in constant time.

With loop unrolling of a loop of size n, all statements in the loop are repeated n-times. This has at best a linear time impact.

Set-Up

We measured the performance of unrolling versus loop invariants for different numbers of loop iterations using two contracts.

The first contract can be seen in Listing 5.2. With the constant value `1`, the number of loop iteration can be set. In this loop, just two additions are performed.

Also, on line 8 of this listing, the loop invariant can be seen. To measure the runs for loop unrolling, the loop invariant was removed.

```
1 l: constant(int128) = 18446744073709551615
2
3 #@ ensures: res == sum(range(l)) + l
4 @public
5 def foo(a: int128):
6     res: int128 = 0
7     for i in range(l):
8         #@ invariant: res == sum(previous(i)) +
    loop_iteration(i)
9         res += i + 1
```

Listing 5.2: A contract with a simple loop that adds up all values from 1 to *l*.

The second contract can be seen in Appendix D.2. This contract also performs some additions inside the loop, but now these calculations introduce more dependencies between loop iterations.

There is an if statement at the end of the contract in the Appendix D.2. Chained conditionals lead to exponentially many program paths, so that symbolic execution, like Silicon, does not scale well. Therefore, in addition to measuring the time with and without the loop invariants, we also measured all runs again without loop invariants and without the if statement in the body of the loop.

However, to handle overflows, some if statements might get implicitly generated by 2VYPER as well. As already explained, Silicon might scale not so well, with such chained conditionals. Therefore, we measured all runs again using Carbon as a back end.

Results

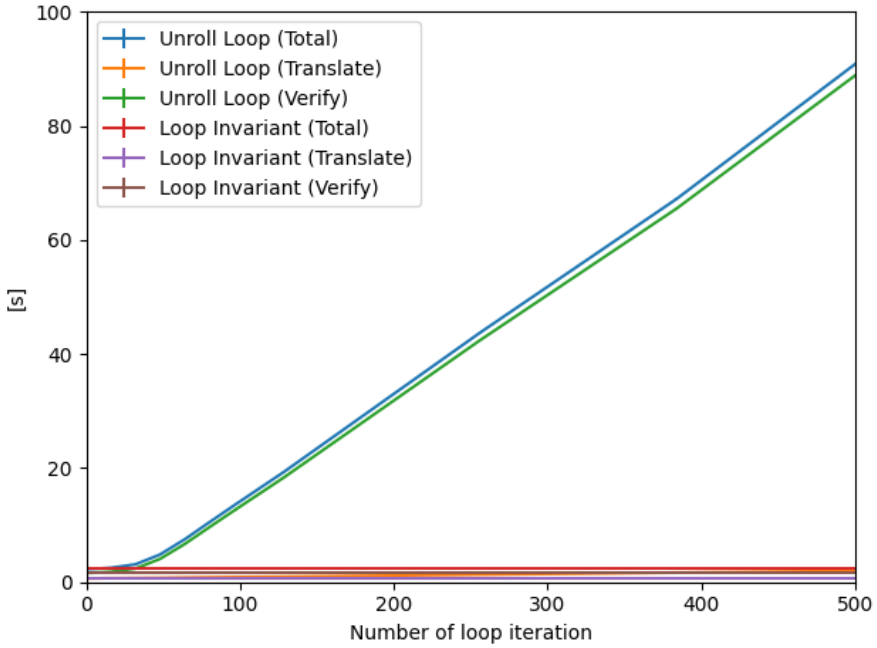
For the first contract, it can be seen in Figure 5.3 that the time remained constant, as expected, when loop invariants are used. Also, with just such a simple loop body, unrolling the loop takes $O(\textit{loop iterations})$ time.

For the second contract, the runs using Silicon as back end can be seen in Figure 5.4. Again, with loop invariants, the time it takes to verify the contract stayed constant.

With the additional conditional at the end of the loop body, Silicon performs worse than without this conditional. But in both cases where the loop gets unrolled, the time is exponentially dependent on the number of loop iterations.

The measurements using Carbon as back end can be seen in Figure 5.5. All phases except the verification stayed the same as in the measurements with Silicon as back end. Therefore, the translation time stayed the same as in

Figure 5.3: Time impact to unroll a simple loop instead to having loop invariants using the Silicon back end.



the measurements using Silicon as back end. The translation time is nearly perfect linear.

With small numbers of loop iterations, Carbon uses almost the same time to verify the contract as it takes time to translate it. With more loop iterations, it can be seen that also Carbon needs more than linear time for the verification. But Carbon clearly outperforms Silicon in this example.

As for Silicon, the verification time of the contract using loop invariants is also independent from the number of loop iterations, when we use Carbon.

5.2.1 Discussion

Previously, all loops got unrolled. We could show that for a simple loop this has at best a linear, but can easily have an exponential performance impact. If there are conditionals in an unrolled loop, we could show that Carbon clearly outperforms Silicon.

With our new loop invariants, we were able to show that the verification time is not dependent on the number of loop iterations any more. Our measurements show that these new loop invariants for Vyper contracts are able to massively outperform the previous method. Verifying a loop with just two or three loop iterations, our new method is already faster than loop unrolling.

Figure 5.4: Time impact to unroll a loop instead of having loop invariants using the Silicon back end, with two versions of the second contract.

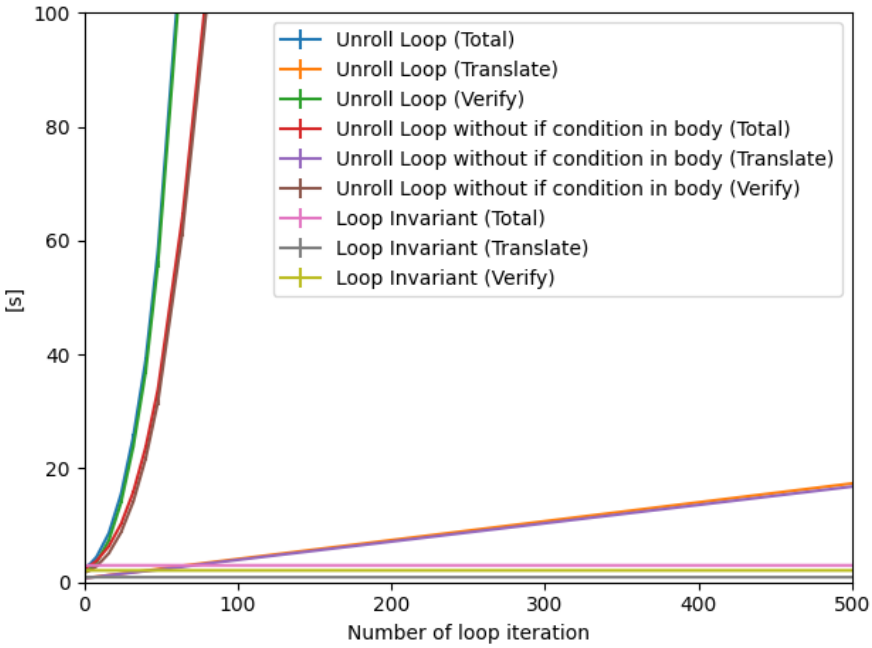
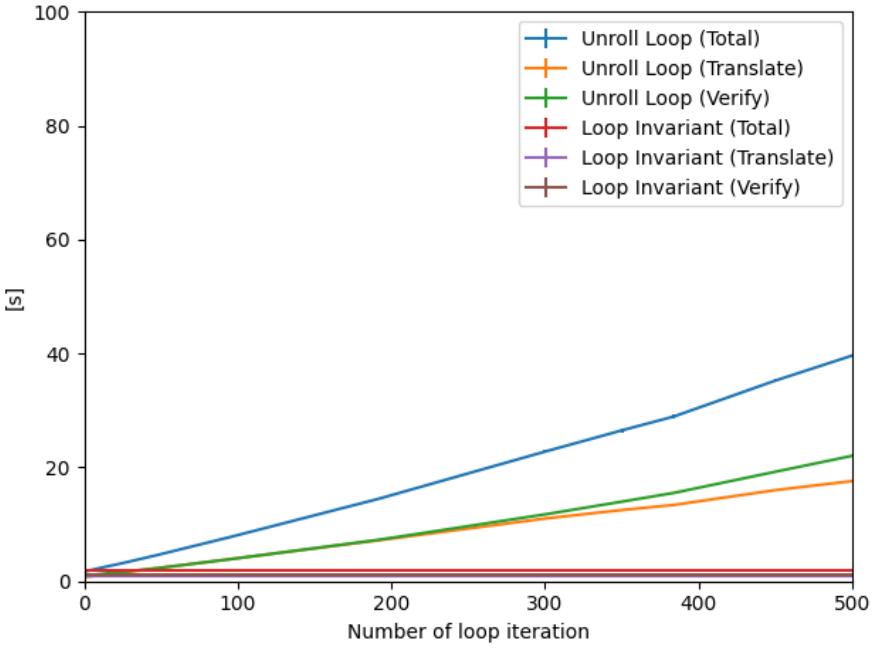


Figure 5.5: Time impact to unroll a loop instead of having loop invariants using the Carbon back end.



5.3 Pure Functions

Pure functions are analysed in two ways.

- **Performance**: Does the encoding scale for complex pure functions?
- **Expressiveness**: Can our design of pure functions be used in real world contracts to capture their property?

5.3.1 Expressiveness

The Uniswap contract has some attributes that are not captured with state variables but are modelled in private, constant functions. We want to show that one can specify the contract's behaviour with respect to these private, constant functions in postconditions.

For the expressiveness part, we analysed Uniswap and applied the pure decorator to pure functions. With these functions, we managed to successfully verify some important properties of Uniswap. In Appendix D.1, parts of the Uniswap contract with specification that model one of Uniswap's important properties can be seen.

With just an annotation overhead of just one pure decorator, it is possible to convert any private, constant function that does not log an event into a pure function.

Uniswap enables trading with tokens. The Ether costs for a token is dependent on the relative supply of the tokens and the Ether of the contract. For example, if a lot of Ether is available but only few tokens, the price for a token would be high.

A property of Uniswap is this dependency of Ether and tokens. This property is modelled using private, constant functions in Uniswap.

This property of Uniswap could not directly be captured previously. Therefore, a postcondition that states that the amount of tokens one would get before a purchase is greater than after a purchase, can only be captured now using pure functions.

A version of this postcondition as a 2VYPER specification for the default function of Uniswap can be seen in Listing 5.3. We could also show that this property holds for all other functions that perform a purchase of tokens.

The idea behind the postcondition is that after the purchase, the token supply decreased and the Ether balance of the contract increased. Therefore, before calling the default function the price of tokens is lower than after calling it.

```
1 #@ ensures: success() ==> forall({token_reserve: uint256},
    token_reserve > 0 ==> \
2     #@ old(result(self.getInputPrice(msg.value, self.balance,
      token_reserve))) \
3     #@ >= result(self.getInputPrice(msg.value, self.balance,
    token_reserve)))
```

Listing 5.3: The postcondition of Uniswap's default function.

5.3.2 Performance

Set-Up

The number of **ensures** clauses that are generated for pure functions is dependent on multiple factors.

According to our encoding in Chapter 3.3.2, each Vyper statement gets translated to a boolean expression that constrains the result struct of the pure function. If this line might revert, another such expression is needed to model this. If this line was inside of a loop- or if-scope, another boolean expression might get generated for every such nested scope. If there is a loop that gets unrolled, all statements inside the loop get encoded to these expressions multiple times.

All of these expressions get their own postcondition in the encoded pure function. Therefore, the number of postconditions for encoded pure functions is in $O(\#nested\ scopes * \#lines\ of\ code * \#iteration\ of\ the\ largest\ loop)$. All of these postconditions could also be combined using **and** operations. Therefore, we call these generated postconditions from now on *conjuncts*.

In Listing 5.4, a contract can be seen with a pure function `foo`. This pure function has a loop that gets unrolled `l`-times. If `l` were 1, 22 conjuncts would get generated to encode `foo`. For `l` being 10, it would be 121 conjuncts. For every further 10 iterations, it takes 110 more conjuncts to encode the function.

We are using this contract to test the impact of the number of conjuncts on the verification performance.

```
1 l: constant(int128) = 80
2
3 #@pure
4 @private
5 @constant
6 def foo(a: bool) -> int128:
7     res: int128 = 0
8     for i in range(l):
```

```
        if a:
            break
        res += i
        continue
        break
    return res


#@ ensures: result(self.foo(False)) == 5
#@ ensures: result(self.foo(True)) == 0
@public
def test(i: int128):
    pass
```

Listing 5.4: A contract with a pure function.

Results

In Figure 5.6, it can be seen that the time it takes to verify the contract is exponentially dependent on the number of generated conjuncts of a pure function.

However, no pure function from a real world contract that we encountered needed more than 1000 conjunctions in the encoding. Even for 1000 conjunctions it takes only roughly 30 seconds to verify the contract.

The pure function `getInputPrice` of the Uniswap contract in Appendix D.1 for example, only needs 22 conjuncts for its encoding.

5.3.3 Discussion

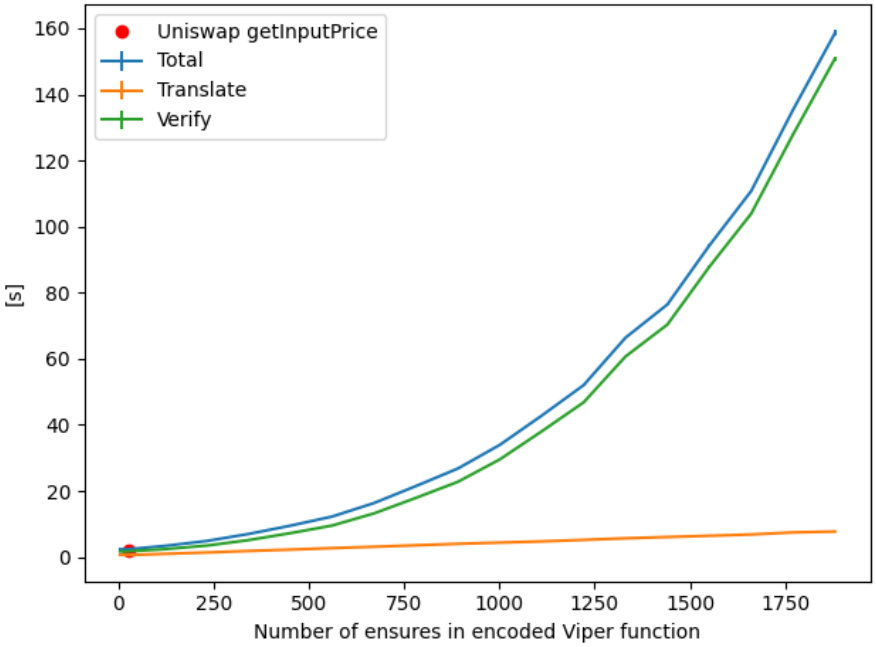
A state variable models one value. Dependencies between values cannot be described using state variables. One example is the dependency between Ether and tokens in the Uniswap contract. So there are further attributes of a contract than just its state variables.

We could show that our implementation of pure functions can be used to express properties about such attributes of a contract, which would have been difficult to express otherwise.

For example, the postcondition of Uniswap's default function would have been impossible to capture without pure functions.

However, pure functions introduce a performance impact. If a lot of postconditions are necessary to encode a pure function, more time is needed to verify the contract. But no pure functions of any real world contract that we checked used more than 1000 postconditions to encode, therefore this performance impact is only in the order of a few seconds.

Figure 5.6: Time impact for pure functions that have different amount of conjuncts in the encoded Viper function, with an example of a real world pure function.



5.4 Nonlinear Integer Arithmetic

We analysed two aspects for nonlinear integer arithmetic.

The first aspect is the performance and stability. We tested the performance and stability difference between 2VYPER with the uninterpreted nonlinear functions and 2VYPER using Z3's interpreted operations on various contracts.

The second aspect is the annotation overhead that lemmas might introduce.

5.4.1 Performance and Stability

Definitions

We defined a *time-out* to be if a verification phase takes more than 2.5 minutes.

We define that 2VYPER can *reliably* verify a property, if 2VYPER always outputs the same verification result for a contract. This is independent from the fact if the specification is actually satisfiable or not.

We say that 2VYPER's verification time is *stable* for a contract if 2VYPER does not time-out and the distribution of all measured runs has a variance of less than nine points.

Set-Up

We evaluated our approach on nonlinear parts of eight real world contracts.

After a few tests runs, we detected that there are three categories of such nonlinear parts, for which 2VYPER with just built-in integers behaves differently.

- **Stable**: There are some contracts with nonlinear arithmetic, where 2VYPER with just built-in integers can reliably verify a property and the verification time for this is stable.
- **Partially Stable**: For some contracts with nonlinear arithmetic, their property could get reliably verified, but the verification time for this was not stable.
- **Unstable**: In the last group of contracts with nonlinear arithmetic, their properties could not be verified reliably and the verification time was not stable either.

Approximately eighty percent of the found nonlinear parts of real world contracts were partially stable. For the rest, only five percent of them were unstable.

We will now show a selection of contracts that were analysed more precisely.

The first such contract can be seen in Listing 5.5. For some small range of x, the function `foo` returns the result of a cubic polynomial. The property that has to get verified is an upper bound for the result of this function.

```
1  # High order poly
2
3  #@ ensures: success() ==> result() <= 63362376
4  #@ ensures: success() ==> x >= 4
5  @public
6  @constant
7  def foo(x:uint256) -> uint256:
8      assert x <= 400
9      #@ lemma_assert interpreted(x * x * x - 4 * x * x + 6 * x
       - 24 <= 63362376)
10     #@ lemma_assert interpreted(6 * x == 4 * x + 2 * x) and
       interpreted(2 * x == x + x)
11     return x * x * x - 4 * x * x + 6 * x - 24
```

Listing 5.5: A function returning the result of a cubic polynomial.

The next contracts are variations of a Newton step. In Listing 5.6 a contract can be seen that performs a Newton step on a quadratic polynomial. As the postconditions of the function `foo` already implies, the root of this polynomial would be two.

The contract in listing 5.7 performs a Newton step on a cubic polynomial. For this polynomial the root would be four.

For some contracts the polynomial or the Newton step is a property of the contract and is used at multiple locations. For this they use private, constant functions. In Listing 5.8 an alternative version of the contract in Listing 5.6 can be seen. Here, the Newton step is extracted to a pure function.

```
# Newton step

#@ lemma_def mul_sign(x: int128):
    #@ x >  0 ==> 1 * x >  0
    #@ x <  0 ==> 1 * x <  0
    #@ x == 0 ==> 1 * x == 0

#@ ensures: success() ==> x >  2 ==> result() <= x
#@ ensures: success() ==> x <  2 ==> result() >= x
@public
@constant
def foo(x: int128) -> int128:
    if x == 2:
        return x

    #@ lemma_assert lemma.mul_sign(x)
    #@ lemma_assert x > 2 ==> x - ((x * x - 4 * x + 4) / (2 *
    x - 4)) <= x
    #@ lemma_assert (x == 1 or x == 0) ==> x - ((x * x - 4 *
    x + 4) / (2 * x - 4)) == 1
    return x - ((x * x - 4 * x + 4) / (2 * x - 4))
```

Listing 5.6: A function performing a Newton step of a quadratic polynomial.

```
# Newton step cube

#@ lemma_def times_4(i: int128):
    #@ i * 2 == i * 1 + i * 1
    #@ i * 4 == i * 2 + i * 2

#@ ensures: success() ==> x > 4 ==> result() <= x
#@ ensures: success() ==> x < 4 ==> result() >= x
@public
@constant
def foo(x: int128) -> int128:
    #@ lemma_assert 3 * x * x - 8 * x + 6 == ((3 * x) - 8) *
    x + 6
```

```
13      #@ lemma_assert lemma.times_4(x) and ((3 * x) - 8) * x +
        6 > 0
14      return x - ((x * x * x - 4 * x * x + 6 * x - 24) / (3 * x
        * x - 8 * x + 6))
```

Listing 5.7: A function performing a Newton step of a cubic polynomial.

```
1  # Newton step pure
2
3  #@ lemma_def mul_sign(x: int128):
4      #@ x >  0 ==> 1 * x >  0
5      #@ x <  0 ==> 1 * x <  0
6      #@ x == 0 ==> 1 * x == 0
7
8  #@pure
9  @private
10 @constant
11 def newtonStep(x: int128) -> int128:
12     if x == 2:
13         return x
14     return x - ((x * x - 4 * x + 4) / (2 * x - 4))
15
16 #@ ensures: success() ==> x >  2 ==> result() <= x
17 #@ ensures: success() ==> x <  2 ==> result() >= x
18 @public
19 @constant
20 def foo(x: int128) -> int128:
21     #@ lemma_assert lemma.mul_sign(x)
22     #@ lemma_assert x > 2 ==> x - ((x * x - 4 * x + 4) / (2 *
        x - 4)) <= x
23     #@ lemma_assert (x == 1 or x == 0) ==> x - ((x * x - 4 *
       x + 4) / (2 * x - 4)) == 1
24     return self.newtonStep(x)
```

Listing 5.8: A private, constant function performing a Newton step of a quadratic polynomial.

The last example of our analysed nonlinear contracts can be seen in Appendix D.3. Here, a calculation similar to a quadratic Newton step is performed in a loop. This is a modified version of Stableswap's `get_y` function.

In all of these contracts, the used lemmas to verify the properties are shown as well. With the exception of the first contract in Listing 5.5, none of the lemmas use an interpreted scope, so that the performance can be measured using only uninterpreted functions for these nonlinear problems. This contract of Listing 5.5 also has a version that does not use an interpreted scope in any of its lemmas. It can be seen in Appendix D.4.

If not stated otherwise, the lemmas were removed while asserting the version without uninterpreted functions.

Results: 2VYPER with Z3's Interpreted Operations

The contract of Listing 5.5 was the only contract for which 2VYPER without uninterpreted functions could reliably and successfully verify the property and had a stable total time.

For all other contracts 2VYPER without uninterpreted functions timed out in some of the runs.

- The "High order poly" contract of listing 5.6 timed out zero times.
  The mean total time was 1.887 seconds with a variance of 0.056.
  The property could be successfully and reliably verified.
- The "Newton step" contract of listing 5.6 timed out 15 times.
  The mean total time was 25.589 seconds with a variance of 30.732.
  The property could be successfully and reliably verified.
- The "Newton step cube" contract of listing 5.7 timed out 5 times.
  The mean total time was 38.934 seconds with a variance of 457.582.
  The property could not be reliably verified, even with lemmas.
- The "Newton step pure" contract of listing 5.8 timed out 21 times.
  The mean total time was 3.596 seconds with a variance of 1.076.
  The property could be successfully and reliably verified.
- The "Stableswap y" contract in the appendix D.3 timed out 4 times.
  The mean total time was 27.714 seconds with a variance of 1086.758.
  The verification of this property reliably failed.

In Figure 5.7, the standardized distributions of the measured total times can be seen. Besides removing the first ten warm-up runs, all runs that timed out were omitted as well. To generate this plot, all remaining measured times were standardized and then a Gaussian kernel estimate was performed.

Only the "High order poly" contract has approximatively a steep normal distribution of the measured times.

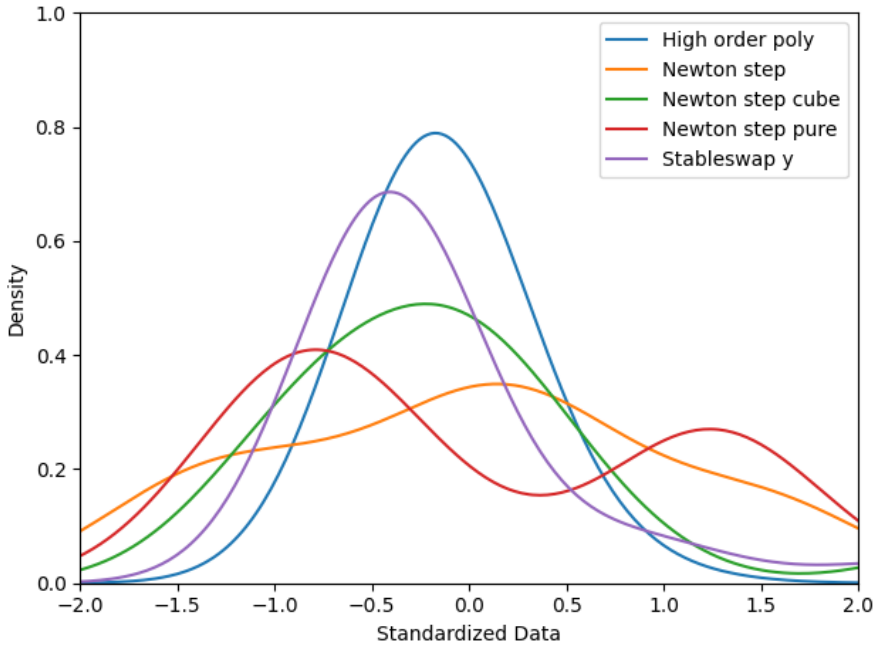
Out of 36 verification attempts that did not time out, the measured verification of "Stableswap y" was around ten seconds 27 times and between 20 and 120 seconds nine times. This can be seen from the steep curve of the normal distribution and a slow decay to the right.

The high variance of the "Newton step cube" can be seen from the flat curve of its distribution.

For the "Newton step" contract, there are times measured around 15, 25 and 30 seconds. This can be seen from the three bumps in its curve.

The "Newton step pure" timed out a lot of times. If it did not time out, it either took two to three seconds or five seconds to verify the property. These two means can be seen in the distribution of the measured runs.

Figure 5.7: The distributions of the measured total times, needed to verify the mentioned contracts, using 2VYPER without wrapped-integers.



Results: 2VYPER with Uninterpreted Functions

As mentioned in the set-up, the contract of Listing 5.5 has two versions of used lemmas. In the listing, the version with interpreted lemmas is shown. In the Appendix D.4, the version without an interpreted scope can be seen.

Using uninterpreted functions, all properties could be reliably and successfully verified. In addition, all of the contracts shown in this section could be verified with a stable time. With the exception of the contract in Appendix D.4, all variances were below 0.1.

Using uninterpreted functions, no measured run took more than 10 seconds for any of the contracts.

- The "High order poly" contract in the appendix D.4 timed out zero times.
  The mean total time was 7.929 seconds with a variance of 2.095.
  The property could be successfully and reliably verified.

- The ”High order poly with interpreted” contract of listing 5.6 timed out zero times.
  The mean total time was 3.286 seconds with a variance of 0.065.
  The property could be successfully and reliably verified.
- The ”Newton step” contract of listing 5.6 timed out zero times.
  The mean total time was 3.349 seconds with a variance of 0.064.
  The property could be successfully and reliably verified.
- The ”Newton step cube” contract of listing 5.7 timed out zero times.
  The mean total time was 4.190 seconds with a variance of 0.060.
  The property could be successfully and reliably verified.
- The ”Newton step pure” contract of listing 5.8 timed out zero times.
  The mean total time was 3.596 seconds with a variance of 0.070.
  The property could be successfully and reliably verified.
- The ”Stableswap y” contract in the appendix D.3 timed out zero times.
  The mean total time was 3.312 seconds with a variance of 0.064.
  The property could be successfully and reliably verified.

Again, we generated the plot in Figure 5.8 using Gaussian kernel estimate. The higher variance of the ”High order poly” contract can be seen from the flat curve of its distribution. All other contracts had approximatively a steep normal distribution of the measured times.

It can be seen, that with a lot of lemmas, as in the ”High order poly” contract, the variance of the measured times increases.

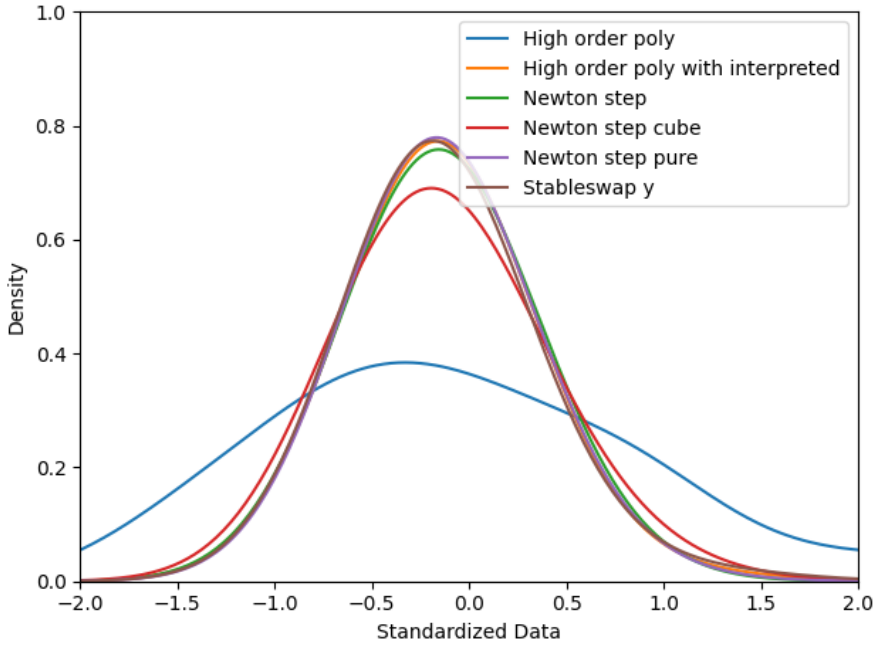
5.4.2 Annotation Overhead

With just two to four lemmas, all properties of the Newton step examples could be verified with uninterpreted functions. The property of the altered version of Stableswap's `get_y` could be verified using just three lemmas. These lemmas capture often just simple mathematical properties like `x` $>$ `0` implies `1` $*$ `x` $>$ `0` or transformations of a nonlinear formula of the Vyper contract.

The only contract that uses more lemmas can be seen in Appendix D.4. A lot of lemmas are needed to capture for example what happens when `400` $*$ `400` gets evaluated.

In this cases, often the interpreted operations perform well using Z3. For example all 23 lemmas of the ”High order poly with interpreted” contract could be rewritten in just the two lemmas of the ”High order poly” contract.

Figure 5.8: The distributions of the measured total times, needed to verify the mentioned contracts, using 2VYPER with wrapped-integers.



5.4.3 Discussion

The results from our measurements show that using uninterpreted functions, the verification is more stable and faster.

The drawback of our method with these functions is the operations with constant values. If there is one, a lot of lemmas are needed.

From our measurements we could identify the following points, that seems to influence, if a property can be stably and reliably verified using Z3:

- The polynomial has no zero-crossing.
- The possible values for all unknown values have a tight bound.
- Operations with an exact / constant value is performed.

If these properties are fulfilled, Z3 seems to be good at verifying a property even with interpreted nonlinear operations.

Since Z3 seems to be good at verifying properties with operations using constant values, as in "High order poly" instead of using a lot of lemmas, the interpreted scope can be used and all needed information can be specified in just a few interpreted lemmas.

Using normal lemmas and interpreted lemmas, both interpreted and uninterpreted nonlinear arithmetic can be used and we enable the best of both

worlds for a contract designer.

5.5 Inter Contract Invariants

To evaluate inter contract invariants, we analysed if such invariants can be successfully verified in real world contracts.

Set-Up

Only very few verifiers for smart contract support inter contract invariants. VerX [27] is one of those that do support them. But VerX does not allow interface abstraction and requires the whole code of every involved contract.

Besides mentioning that they could verify all collected inter contract invariants, they also published all their used Solidity contracts along with the properties to verify [15].

The inter contract invariants examples are in their ICO and Mana examples.

To test our implementation, we translated their Solidity contracts to Vyper contracts, generated interfaces for some contracts and tried to verify the same properties.

Owner property

There are multiple inter contract properties in these two examples. We could successfully verify all of them. Since the analysis for these properties would be similar, we focus here only on the example that VerX also used in their paper.

```
always((MANAContinuousSale.started != true)==> MANAToken.
    owner == MANACrowdsale)
```

These three names MANAContinuousSale, MANAToken and MANACrowdsale are three contracts. VerX needs total knowledge about all involved contracts. Our implementation of inter contract invariants only needs one contract as the main contract and for the other contracts only their interface has to be provided. The Mana crowd sale contract is the one contract that manages the others. Therefore, we verified the Mana crowd sale contract using interface abstractions for the other two contracts and then verified that the other two contracts implement their interfaces.

In Listing 5.9 part of the Mana token interface can be seen. The token models its owner using an address. Only this address can again change this owner-address. We encode this fact using a conditional caller private expression, which can be seen on line 12 of this listing.

```
# Mana token interface

# [Some imports]

#@ interface

#@ ghost:
    #@ def token_owner() -> address: ...
    # [Some further ghost functions]

# token_owner is only modifiable by the token_owner himself
#@ caller private: conditional(token_owner(self) == caller(),
    token_owner(self))


#@ ensures: msg.sender != old(token_owner(self)) ==> revert()
#@ ensures: success() ==> token_owner(self) == newOwner
@public
def transferOwnership(newOwner: address):
    raise "Not implemented"

# [Some more public function declarations]
```

Listing 5.9: Part of the Mana token interface.

In Listing 5.10 part of the Mana continuous sale interface can be seen. This contract has also an owner-address. There is also a flag that models if the sale has already started. As the conditional caller private on line 20 of this listing already gives away, only the owner can start the sale.

```
# Mana continuous sale interface

# [Some imports]

#@ interface

#@ ghost:
    #@ def started() -> bool: ...
    #@ def owner() -> address: ...
    # [Some further ghost functions]

# The owner stays the same
#@ invariant: owner(self) == old(owner(self))
# Once started, it stays started
```

```
15 #@ invariant: old(started(self)) ==> started(self)
16
17 # [More global specification]
18
19 # started flag is only modifiable by owner
20 #@ caller private: conditional(owner(self) == caller(),
       started(self))
21
22
23 #@ ensures: success() ==> result() == started(self)
24 @public
25 @constant
26 def is_started() -> bool:
27     raise "Not implemented"
28
29
30 #@ ensures: msg.sender != owner(self) ==> revert()
31 #@ ensures: old(started(self)) ==> revert()
32 #@ ensures: success() ==> started(self)
33 @public
34 def start():
35     raise "Not implemented"
36
37 # [Some more public function declarations]
```

Listing 5.10: Part of the Mana continuous sale interface.

Parts of the Mana crowd sale contract can be seen in the Listing 5.11. For this example all other functions except `beginContinuousSale` are omitted. The property we want to encode from VerX can be seen on line 14 of this listing. The original property from VerX would be violated during the execution of the `beginContinuousSale` function. Since we cannot atomically transfer the ownership of the token and start the continuous sale, we prepend **`not`** `locked("lock")` to the property from VerX.

```
1 #@ config: trust_casts
2
3 from . import mana_continuous_sale_interface
4 from . import mana_token_interface
5
6 token: mana_token_interface
7 continuousSale: mana_continuous_sale_interface
8 # [Some more state variables]
9
10 #@ invariant: self.token == old(self.token)
```

```
11 #@ invariant: self.continuousSale == old(self.continuousSale)
12
13 #@ inter contract invariant: owner(self.continuousSale) ==
       self
14 #@ inter contract invariant: not locked("lock") ==> not
       started(self.continuousSale) ==> token_owner(self.token)
       == self
15 # [Further contract specifications]
16
17
18 @nonreentrant('lock')
19 @public
20 @payable
21 def beginContinuousSale():
22     assert not self.continuousSale.is_started()
23     self.token.transferOwnership(self.continuousSale)
24     self.continuousSale.start()
25
26 # [Some more functions]
```

Listing 5.11: Part of the Mana crowd sale contract.

5.5.1 Discussion

With just two caller private expressions and a few invariants, we can verify the inter contract invariant which VerX used in their paper.

In addition to this example, we could verify all inter contract invariants in the contracts used by VerX.

This shows that our method is powerful enough that real world inter contract invariants can be modelled and verified.

Chapter 6

Conclusion and Future Work

In this thesis we extended 2VYPER with new features that enable the verification of advanced properties of real world contracts.

To verify these advanced properties, two of our features improved the performance of 2VYPER, by introducing a way to prevent in-lining private function calls and also by enabling an alternative to loop unrolling for Vyper contracts.

Both of these features introduce new possibilities to write specification for a contract that was previously not possible. Loop invariants can now be specified and enable verification of loops in constant time. We introduced specification for private functions for the prevention of in-lining calls to those private functions.

We could show that if a Vyper contract contains nonlinear arithmetic, the verification of the specification can be unstable and unreliable. We introduced uninterpreted functions to prevent this. With uninterpreted functions, additional user input is needed. For this, we introduced lemmas in 2VYPER. We showed that in some cases, a lot of lemmas would be needed and that in this cases often the interpreted nonlinear operations of the SMT solver are stable and fast. For this, we give a user the possibility to use again the interpreted version of the nonlinear operations with the interpreted lemmas. With this, a user has the best of both worlds.

Further we enabled inter contract invariants for 2VYPER. This gives contract designers a tool to specify invariants that capture properties of multiple contracts. Our design of inter contract invariants requires only from the contract that gets verified its full code, for all other contracts only their interfaces is required.

2VYPER is a sound verifier with many features. But there are certainly further things to do.

We were able to improve the performance of 2VYPER but certainly it is still possible to even further improve its performance.

During this thesis, a new version of Vyper was released. 2VYPER should be extended to also support this new release.

2VYPER has a unique way of handling resources of Vyper contracts. This resource handling is currently only supported for a single contract. This feature could be extended for multiple contracts.

Appendix A

Encoding of Loops in Pure Functions

```python
#@pure
@private
@constant
def foo(a: bool) -> int128:
    res: int128 = 0
    for i in range(42):
        #@ invariant: res == i
        #@ invariant: a ==> loop_iteration(i) == 0
        if a:
            break
        res += 1
    return res
```

Listing A.1: A pure function with a loop and a break statement.

```
function p$foo(self: Struct, a: Bool): Struct
  // [Some previous postconditions]
  // First loop invariant
  ensures (struct_get(result, 10): Bool) ==
    ((struct_get(result, 9): Int)) ==
      (struct_get(result, 8): Int)
  // Second loop invariant
  ensures (struct_get(result, 11): Bool) ==
    ((struct_get(result, 2): Bool) ==>
      (struct_get(result, 6): Int) == 0)
  // Store if-condition
  ensures (struct_get(result, 10): Bool)
    && (struct_get(result, 11): Bool) ==>
      (struct_get(result, 12): Bool) ==
        (struct_get(result, 2): Bool)
  // [Some checks for overflow]
```

```
17  // Increment res
18  ensures (struct_get(result, 10): Bool) &&
19    (struct_get(result, 11): Bool) &&
20      !(struct_get(result, 13): Bool) ==>
21        (struct_get(result, 15): Int)) ==
22        (struct_get(result, 9): Int + 1
23  // Increment the loop index
24  ensures (struct_get(result, 16): Int) ==
25    (struct_get(result, 6): Int) + 1
26  // Condition to break out of the loop if the end was
     reached
27  ensures (struct_get(result, 17): Bool) ==
28    ((struct_get(result, 16): Int) == 42)
29  // Exclusive or the conditions how to break out of a loop
30  ensures (struct_get(result, 10): Bool) &&
31    (struct_get(result, 11): Bool) &&
32    (struct_get(result, 12): Bool) &&
33    !((struct_get(result, 10): Bool) &&
34      (struct_get(result, 11): Bool) &&
35      (struct_get(result, 17): Bool)) ||
36    !((struct_get(result, 10): Bool) &&
37      (struct_get(result, 11): Bool) &&
38      (struct_get(result, 12): Bool)) &&
39    ((struct_get(result, 10): Bool) &&
40      (struct_get(result, 11): Bool) &&
41      (struct_get(result, 17): Bool))
42  // Merge the different versions of the res variables
43  // First the break out of the loop at the end of the loop
44  ensures (struct_get(result, 18): Int) ==
45    ((struct_get(result, 10): Bool) &&
46      (struct_get(result, 11): Bool) &&
47      (struct_get(result, 17): Bool) ?
48        (struct_get(result, 15): Int) :
49        (struct_get(result, 9): Int))
50  // Second merge, because of the explicit break statement
51  ensures (struct_get(result, 19): Int) ==
52    ((struct_get(result, 10): Bool) &&
53      (struct_get(result, 11): Bool) &&
54      (struct_get(result, 12): Bool) ?
55        (struct_get(result, 9): Int) :
56        (struct_get(result, 18): Int))
57 // [Some later postconditions]
```

Listing A.2: Parts of an encoded pure Vyper function with a loop and a break statement.

Appendix B

Wrapped Integer Domain

```
/*
  Copyright (c) 2020 ETH Zurich
  This Source Code Form is subject to the terms of the Mozilla Public
  License, v. 2.0. If a copy of the MPL was not distributed with this
  file, You can obtain one at http://mozilla.org/MPL/2.0/.
*/

domain $Int {
  function $wrap(x: Int): $Int
  function $unwrap(x: $Int): Int

  axiom $wrap_ax {
    forall i: Int :: {$wrap(i)} $unwrap($wrap(i)) == i
  }

  axiom $unwrap_ax {
    forall i: $Int :: {$wrap($unwrap(i))} $wrap($unwrap(i)) == i
  }

  // Operations on the wrapped integer

  function $w_mul(x: $Int, y: $Int): $Int

  function $w_mulI(x: $Int, y: $Int): $Int // an intermediate (also limited) mul function

  function $w_mulL(x: $Int, y: $Int): $Int // a limited mul function

  // The mulitplication is commutative. Therefore, to provide good expressiveness,
  // all axioms e.g. distributivity should be provided for both operands.
  //  -> (a * (b + c)) == (a * b) + (a * c)
  //  -> ((b + c) * a) == (b * a) + (c * a)
  // To reduce the number of axioms, we introduced the intermediate mul function.
```

```
33
34  function $w_abs(x: $Int): Int
35
36  function $w_mod($a: $Int, $b: $Int): $Int
37
38  function $w_modL($a: $Int, $b: $Int): $Int // a limited mod
      function
39
40  function $w_div($a: $Int, $b: $Int): $Int
41
42  function $w_div_down($a: $Int, $b: $Int): $Int
43
44  function $w_div_nat($a: $Int, $b: $Int): $Int
45
46  function $w_div_natL($a: $Int, $b: $Int): $Int // a limited
      div_nat function
47
48  // All axioms marked with "tested" were once automatically
      converted into a method with an assert and checked that
49  // the axioms hold on their own.
50  // Some axioms for the modulus (marked with "tested (only
      for natural numbers)") were only tested for natural
      numbers,
51  // since the Vyper modulus is a "remainder" operation but
      Viper uses the "true" mathematical modulus. Both moduli
52  // behave the same on natural numbers, but they diverge on
      negative inputs.
53
54  // The converted methods used "function f_abs(x: Int): Int
      {i < 0 ? -i : i}", the standard multiplication, modulus
55  // and division of Viper.
56  // For the conversion, all £unwrap and £wrap were removed,
      £w_abs got replaced with f_abs.
57  // £w_mul, £w_mulI, £w_mulL were replaced by the standard
      multiplication.
58  // £w_mod, £w_modL were replaced by the standard modulus.
59  // £w_div, £w_div_down, £w_div_nat, £w_div_natL were
      replaced by the standard division.
60
61  // tested
62  axiom $w_abs_ax_1 {
63    forall i: $Int :: {$w_abs(i)} $unwrap(i) < 0 ==> $w_abs(i
      ) == -$unwrap(i)
64  }
65
66  // tested
67  axiom $w_abs_ax_2 {
68    forall i: $Int :: {$w_abs(i)} $unwrap(i) >= 0 ==> $w_abs(
      i) == $unwrap(i)
69  }
70
71  // tested
72  axiom $w_mul_intermediate {
73    forall i: $Int, j: $Int :: {$w_mul(i, j)} $w_mul(i, j) ==
      $w_mulI(i, j)
74  }
75
76  // tested
77  axiom $w_mul_limited {
```

```
78      forall i: $Int, j: $Int :: {$w_mul(i, j)} $w_mul(i, j) ==
        $w_mulL(i, j)
79    }
80
81    // tested
82    axiom $w_mul_intermediate_to_limited {
83      forall i: $Int, j: $Int :: {$w_mulI(i, j)} $w_mulI(i, j)
       == $w_mulL(i, j)
84    }
85
86    // tested
87    axiom $w_mul_commutative {
88      forall i: $Int, j: $Int :: {$w_mul(i, j)} $w_mul(i, j) ==
        $w_mulI(j, i)
89    }
90
91    // tested
92    axiom $w_mul_associative {
93      forall i: $Int, j: $Int, k: $Int :: {$w_mulI(i, $w_mulI(j
       , k))} $w_mulI(i, $w_mulI(j, k)) == $w_mulL($w_mulL(i, j),
        k)
94    }
95
96    // tested
97    axiom $w_mul_distributive {
98      forall i: $Int, j: $Int, k: $Int, l: $Int :: {$w_mulI(i,
       j), $w_mulI(i, k), $w_mulI(i, l)} $unwrap(j) == $unwrap(k)
        + $unwrap(l) ==> $w_mulI(i, j) == $wrap($unwrap($w_mulL(i
       , k)) + $unwrap($w_mulL(i, l)))
99    }
100
101   // tested
102   axiom $w_mul_basic_sign_1 {
103     forall i: $Int, j: $Int :: {$w_mulI(i, j)} $w_mulI(i, j)
       == $w_mulL($wrap(-$unwrap(i)), $wrap(-$unwrap(j)))
104   }
105
106   // tested
107   axiom $w_mul_basic_sign_2 {
108     forall i: $Int, j: $Int :: {$w_mulI(i, j)} $w_mulI(i, j)
       == $wrap(-$unwrap($w_mulL($wrap(-$unwrap(i)), j)))
109   }
110
111   // tested
112   axiom $w_mul_basic_zero_1 {
113     forall i: $Int, j: $Int :: {$w_mulI(i, j)} ($unwrap(i) ==
        0 || $unwrap(j) == 0) ==> $unwrap($w_mulI(i, j)) == 0
114   }
115
116   // tested
117   axiom $w_mul_basic_zero_2 {
118     forall i: $Int, j: $Int :: {$w_mulI(i, j)} (($unwrap(i) >
        0 && $unwrap(j) > 0) || ($unwrap(i) < 0 && $unwrap(j) <
       0)) ==> $unwrap($w_mulI(i, j)) > 0
119   }
120
121   // tested
122   axiom $w_mul_basic_neutral {
```

```
123     forall i: $Int, j: $Int :: {$w_mulI(i, j)} ($unwrap(i) ==
        1 || $unwrap(j) == 0) ==> $w_mulI(i, j) == j
124   }
125
126   // tested
127   axiom $w_mul_basic_proportional {
128     forall i: $Int, j: $Int :: {$w_mulI(i, j)} ($w_abs(
        $w_mulI(i, j)) >= $w_abs(j)) <==> ($w_abs(i) >= 1 ||
        $unwrap(j) == 0)
129   }
130
131   // tested
132   axiom $w_mul_order_1 {
133     forall i: $Int, j: $Int, k: $Int, l: $Int :: {$w_mulI(i,
        $w_mulI(j, l)), $w_mulI(k, l)} ($unwrap($w_mulI(i, j)) >
        $unwrap(k) && $unwrap(l) > 0) ==> $unwrap($w_mulL(i,
        $w_mulL(j, l))) > $unwrap($w_mulI(k, l))
134   }
135
136   // tested
137   axiom $w_mul_order_2 {
138     forall i: $Int, j: $Int, k: $Int, l: $Int :: {$w_mulI(i,
        $w_mulI(j, l)), $w_mulI(k, l)} ($unwrap($w_mulI(i, j)) >=
        $unwrap(k) && $unwrap(l) > 0) ==> $unwrap($w_mulL(i,
        $w_mulL(j, l))) >= $unwrap($w_mulI(k, l))
139   }
140
141   // tested
142   axiom $w_mul_order_3 {
143     forall i: $Int, j: $Int, k: $Int, l: $Int :: {$w_mulI(i,
        $w_mulI(j, l)), $w_mulI(k, l)} ($unwrap($w_mulI(i, j)) >
        $unwrap(k) && $unwrap(l) < 0) ==> $unwrap($w_mulI(k, l)) >
         $unwrap($w_mulL(i, $w_mulL(j, l)))
144   }
145
146   // tested
147   axiom $w_mul_order_4 {
148     forall i: $Int, j: $Int, k: $Int, l: $Int :: {$w_mulI(i,
        $w_mulI(j, l)), $w_mulI(k, l)} ($unwrap($w_mulI(i, j)) >=
        $unwrap(k) && $unwrap(l) < 0) ==> $unwrap($w_mulI(k, l))
        >= $unwrap($w_mulL(i, $w_mulL(j, l)))
149   }
150
151   // tested
152   axiom $w_mul_monotonicity_1 {
153     forall i: $Int, j: $Int, k: $Int, l: $Int :: {$w_mulI(i,
        k), $w_mulI(j, l)} ($w_abs(i) <= $w_abs(j) && $w_abs(k) <=
         $w_abs(l)) ==> $w_abs($w_mulI(i, k)) <= $w_abs($w_mulI(j,
         l))
154   }
155
156   // tested
157   axiom $w_mul_monotonicity_2 {
158     forall i: $Int, j: $Int, k: $Int, l: $Int :: {$w_mulI(i,
        k), $w_mulI(j, l)} ($w_abs(i) < $w_abs(j) && $w_abs(k) <=
        $w_abs(l) && $unwrap(l) != 0) ==> $w_abs($w_mulI(i, k)) <
        $w_abs($w_mulI(j, l))
159   }
160
```

```
161   // tested
162   axiom $w_mul_monotonicity_3 {
163     forall i: $Int, j: $Int, k: $Int, l: $Int :: {$w_mulI(i,
      k), $w_mulI(j, l)} ($w_abs(i) <= $w_abs(j) && $w_abs(k) <
      $w_abs(l) && $unwrap(j) != 0) ==> $w_abs($w_mulI(i, k)) <
      $w_abs($w_mulI(j, l))
164   }
165
166   // tested
167   axiom $w_mod_limited {
168     forall i: $Int, j: $Int  :: {$w_mod(i, j)} $w_mod(i, j)
      == $w_modL(i, j)
169   }
170
171   // tested
172   axiom $w_mod_identity {
173     forall i: $Int, j: $Int :: {$w_mod(i, j)} j != $wrap(0)
      ==> (i == j || i == $wrap(0)) ==> $w_mod(i, j) == $wrap(0)
174   }
175
176   // tested (only for natural numbers)
177   axiom $w_mod_basic_1 {
178     forall i: $Int, j: $Int, l: $Int :: {$w_mod(i, j), $w_mod
      (l, j)}  j != $wrap(0) ==> ($unwrap(i) == $unwrap(l) +
      $w_abs(j) && ($unwrap(l) >= 0 || $unwrap(i) < 0)) ==>
      $w_mod(i, j) == $w_modL(l, j)
179   }
180
181   // tested (only for natural numbers)
182   axiom $w_mod_basic_2 {
183     forall i: $Int, j: $Int, l: $Int :: {$w_mod(i, j), $w_mod
      (l, j)}  j != $wrap(0) ==> ($unwrap(i) == $unwrap(l) -
      $w_abs(j) && ($unwrap(l) <= 0 || $unwrap(i) > 0)) ==>
      $w_mod(i, j) == $w_modL(l, j)
184   }
185
186   // tested (only for natural numbers)
187   axiom $w_mod_basic_3 {
188     forall i: $Int, j: $Int :: {$w_mod(i, j)}  j != $wrap(0)
      ==> (0 <= $w_abs(i) &&  $w_abs(i) < $w_abs(j)) ==> $w_mod(
      i, j) == i
189   }
190
191   // tested
192   axiom $w_mod_basic_4 {
193     forall i: $Int, j: $Int :: {$w_mod(i, j)}  j != $wrap(0)
      ==> $w_abs($w_mod(i, j)) < $w_abs(j)
194   }
195
196   // tested (only for natural numbers)
197   axiom $w_mod_sign_1 {
198     forall i: $Int, j: $Int :: {$w_mod(i, j)} j != $wrap(0)
      ==> ($sign($unwrap($w_mod(i, j))) == $sign($unwrap(i)) ||
      $sign($unwrap($w_mod(i, j))) == 0)
199   }
200
201   // This is only true for a remainder modulus and could not
      be tested.
202   axiom $w_mod_sign_2 {
```

```
203     forall i: $Int, j: $Int, k: $Int :: {$w_mod(i, j), $w_mod
        (k, j)} j != $wrap(0) ==> $unwrap(i) == -$unwrap(k) ==>
        $w_mod(i, j) == $wrap(-$unwrap($w_modL(k, j)))
204   }
205
206   // This is only true for a remainder modulus and could not
        be tested.
207   axiom $w_mod_sign_3 {
208     forall i: $Int, j: $Int :: {$w_mod(i, j)} j != $wrap(0)
        ==> ($w_mod(i, j) == $w_mod(i, $wrap(-$unwrap(j))))
209   }
210
211   // tested
212   axiom $w_mod_mod {
213     forall i: $Int, j: $Int :: {$w_mod(i, j)} j != $wrap(0)
        ==> $w_mod(i, j) == $w_modL($w_modL(i, j), j)
214   }
215
216   // tested (only for natural numbers)
217   axiom $w_mod_decrease {
218     forall i: $Int, j: $Int :: {$w_mod(i, j)} j != $wrap(0)
        ==> $w_abs($w_mod(i, j)) <= $w_abs(i)
219   }
220
221   // This axiom is useful if we have a sum in a modulo
        operation (e.g. ((k + l) % j)) and we want to split the
222   // modulus into multiple terms (e.g. (k % j) and (l % j)).
223   // There is a case distinction first, if (k + l) and ((k %
        j) + (l % j)) is >= 0 or both <= 0 then we have to check:
224   // - if ((k % j) + (l % j)) is >= |j| then we have to
        subtract |j| meaning ((k + l) % j) == (k % j) + (l % j) -
        |j|
225   // - if ((k % j) + (l % j)) is <= -|j| then we have to add
        |j| meaning     ((k + l) % j) == (k % j) + (l % j) + |j|
226   // - else ((k + l) % j) == (k % j) + (l % j)
227   // If one of (k + l) and ((k % j) + (l % j)) is negative
        and the other positive we have to check:
228   // - if ((k % j) + (l % j)) > 0 then we have to subtract |j
        | meaning ((k + l) % j) == (k % j) + (l % j) - |j|
229   //   since the original addition (k + l) was negative and
        the result must be therefore negative.
230   // - if ((k % j) + (l % j)) < 0 then we have to subtract |j
        | meaning ((k + l) % j) == (k % j) + (l % j) + |j|
231   //   since the original addition (k + l) was positive and
        the result must be therefore positive.
232   // This is only true for a remainder modulus and could not
        be tested.
233   axiom $w_mod_add {
234     forall i: $Int, j: $Int, k: $Int, l: $Int :: {$w_mod(i, j
        ), $w_mod(k, j), $w_mod(l, j)} j != $wrap(0) ==> ($unwrap(
        i) == $unwrap(k) + $unwrap(l) ==> (
235       (($unwrap(i) >= 0 && $unwrap($w_modL(k, j)) + $unwrap(
        $w_modL(l, j)) >= 0) || ($unwrap(i) <= 0 && $unwrap(
        $w_modL(k, j)) + $unwrap($w_modL(l, j)) <= 0)) ==> (
236           ($w_abs(j)      <= ($unwrap($w_modL(k, j)) +
        $unwrap($w_modL(l, j))) && ($unwrap($w_modL(k, j)) +
        $unwrap($w_modL(l, j))) <  2 * $w_abs(j) && $w_mod(i, j)
        == $wrap($unwrap($w_modL(k, j)) + $unwrap($w_modL(l, j)) -
         $w_abs(j)))
```

```
237         || (-$w_abs(j)     <  ($unwrap($w_modL(k, j)) +
    $unwrap($w_modL(l, j))) && ($unwrap($w_modL(k, j)) +
    $unwrap($w_modL(l, j))) <  $w_abs(j)     && $w_mod(i, j)
    == $wrap($unwrap($w_modL(k, j)) + $unwrap($w_modL(l, j))))
238         || (-2 * $w_abs(j) <  ($unwrap($w_modL(k, j)) +
    $unwrap($w_modL(l, j))) && ($unwrap($w_modL(k, j)) +
    $unwrap($w_modL(l, j))) <= -$w_abs(j)    && $w_mod(i, j)
    == $wrap($unwrap($w_modL(k, j)) + $unwrap($w_modL(l, j)) +
     $w_abs(j)))
239       )) && (((($unwrap(i) > 0 && $unwrap($w_modL(k, j)) +
    $unwrap($w_modL(l, j)) < 0) || ($unwrap(i) < 0 && $unwrap(
    $w_modL(k, j)) + $unwrap($w_modL(l, j)) > 0)) ==> (
240           (0              <  ($unwrap($w_modL(k, j)) +
    $unwrap($w_modL(l, j))) && ($unwrap($w_modL(k, j)) +
    $unwrap($w_modL(l, j))) <  $w_abs(j)     && $w_mod(i, j)
    == $wrap($unwrap($w_modL(k, j)) + $unwrap($w_modL(l, j)) -
     $w_abs(j)))
241         || (-$w_abs(j)     <  ($unwrap($w_modL(k, j)) +
    $unwrap($w_modL(l, j))) && ($unwrap($w_modL(k, j)) +
    $unwrap($w_modL(l, j))) <  0             && $w_mod(i, j)
    == $wrap($unwrap($w_modL(k, j)) + $unwrap($w_modL(l, j)) +
     $w_abs(j)))
242       ))
243     ))
244   }
245
246   // This could only be tested for an "i" in [-512, 529]. For
        an "i" smaller than -512, the verification took more than
247   // 2 minutes on an Intel Core i9 9900K. For an "i" greater
        than 529, Viper could not verify the expression.
248   axiom $w_mod_mul_basic {
249     forall i: $Int, j: $Int :: {$w_mod($w_mul(i, j), j)} j !=
      $wrap(0) ==> $w_mod($w_mul(i, j), j) == $wrap(0)
250   }
251
252   // If we have a modulus (i % j) multiplied by some k and
        take the modulus of this by j (e.g. ((i % j) * k) % j)
        then
253   // we can simplify the term to (i * k) % j.
254   // This axiom is an adapted version of the lemma "
        lemma_mul_mod_noop_general" from IronClad.
255   // (https://github.com/microsoft/Ironclad/blob/master/
        ironfleet/src/Dafny/Libraries/Math/div.i.dfy)
256   axiom $w_mod_mul_mod_noop {
257     forall i: $Int, j: $Int, k: $Int :: {$w_mod($w_mulI(i, k)
      , j)} j != $wrap(0) ==> (
258           $w_mod($w_mulI(i, k), j) == $w_modL($w_mulL($w_modL(
      i, j), k), j)
259        && $w_mod($w_mulI(i, k), j) == $w_modL($w_mulL(i,
      $w_modL(k, j)), j)
260        && $w_mod($w_mulI(i, k), j) == $w_modL($w_mulL($w_modL(
      i, j), $w_modL(k, j)), j)
261     )
262   }
263
264   // tested (only for natural numbers)
265   axiom $w_mod_mul_vanish {
266     forall i: $Int, j: $Int, k: $Int :: {$w_mod(i, j),
      $w_mulI(k, j)} j != $wrap(0) ==> $w_mod(i, j) == $w_modL(
```

```
    $wrap($unwrap($w_mulL(k, j)) + $unwrap(i)), j)
267 }
268
269 // tested
270 axiom $w_div_div_down{
271   forall i: $Int, j: $Int :: {$w_div(i, j)} $unwrap(j) != 0
    ==> $w_div(i, j) == ($unwrap(i) >= 0 ? $w_div_down(i, j)
    : $wrap(-$unwrap($w_div_down($wrap(-$unwrap(i)), j))))
272 }
273
274 // This is only true for a division with rounding towards
    zero and could not be tested.
275 axiom $w_div_down_div_nat{
276   forall i: $Int, j: $Int :: {$w_div_down(i, j)}
    $w_div_down(i, j) == ($unwrap(j) >= 0 ? $w_div_nat(i, j) :
    $wrap(-$unwrap($w_div_nat(i, $wrap(-$unwrap(j))))))
277 }
278
279 // tested
280 axiom $w_div_nat_limited {
281   forall i: $Int, j: $Int :: {$w_div_nat(i, j)} $w_div_nat(
    i, j) == $w_div_natL(i, j)
282 }
283
284 // tested
285 axiom $w_div_nat_neutral {
286   forall i: $Int, j: $Int :: {$w_div_nat(i, j)} ($unwrap(j)
    == 1 || $unwrap(i) == 0) ==> $w_div_nat(i, j) == i
287 }
288
289 // tested
290 axiom $w_div_nat_self {
291   forall i: $Int :: {$w_div_nat(i, i)} $unwrap(i) > 0 ==>
    $w_div_nat(i, i) == $wrap(1)
292 }
293
294 // tested
295 axiom $w_div_nat_small {
296   forall i: $Int, j: $Int :: {$w_div_nat(i, j)} $unwrap(i)
    >= 0 && $unwrap(j) > 0 ==> ($unwrap(i) < $unwrap(j) <==>
    $w_div_nat(i, j) == $wrap(0))
297 }
298
299 // The verification ran into a timeout.
300 axiom $w_div_nat_dividend_add {
301   forall i: $Int, j: $Int, k: $Int, l: $Int :: {$w_div_nat(
    i, j), $w_div_nat(k, j), $w_div_nat(l, j)} ($unwrap(i) >=
    0 && $unwrap(j) > 0 && $unwrap(k) >= 0 && $unwrap(l) >= 0)
    ==> (
302     $unwrap(i) == $unwrap(k) + $unwrap(l) ==> (
303       (0 <= $unwrap($w_mod(k, j)) + $unwrap(
    $w_mod(l, j)) && $unwrap($w_mod(k, j)) + $unwrap($w_mod(l,
    j)) < $unwrap(j) && $w_div_nat(i, j) == $wrap($unwrap
    ($w_div_natL(k, j)) + $unwrap($w_div_natL(l, j))))
304       || ($unwrap(j) <= $unwrap($w_mod(k, j)) + $unwrap(
    $w_mod(l, j)) && $unwrap($w_mod(k, j)) + $unwrap($w_mod(l,
    j)) < 2 * $unwrap(j) && $w_div_nat(i, j) == $wrap($unwrap
    ($w_div_natL(k, j)) + $unwrap($w_div_natL(l, j)) + 1))
305     ))
```

```
306   }
307
308   // tested
309   axiom $w_div_nat_ordered_by_dividend {
310     forall i: $Int, j: $Int, k: $Int :: {$w_div_nat(i, j),
      $w_div_nat(k, j)} ($unwrap(i) >= 0 && $unwrap(j) > 0 &&
      $unwrap(k) >= 0) ==> (
311       $unwrap(i) <= $unwrap(k) ==> $unwrap($w_div_nat(i, j))
      <= $unwrap($w_div_natL(k, j))
312     )
313   }
314
315   // tested
316   axiom $w_div_nat_ordered_by_divisor {
317     forall i: $Int, j: $Int, k: $Int :: {$w_div_nat(i, j),
      $w_div_nat(i, k)} ($unwrap(i) >= 0 && $unwrap(j) > 0 &&
      $unwrap(k) > 0) ==> (
318       $unwrap(j) <= $unwrap(k) ==> $unwrap($w_div_nat(i, j))
      >= $unwrap($w_div_natL(i, k))
319     )
320   }
321
322   // This can only be verified if we give an upper bound for
      "i".
323   // tested (with i < 2 ** 256)
324   axiom $w_div_nat_decrease {
325     forall i: $Int, j: $Int :: {$w_div_nat(i, j)} ($unwrap(i)
      > 0 && $unwrap(j) > 1) ==> (
326       $unwrap($w_div_nat(i, j)) < $unwrap(i)
327     )
328   }
329
330   // This can only be verified if we give an upper bound for
      "i".
331   // tested (with i < 2 ** 256)
332   axiom $w_div_nat_nonincrease {
333     forall i: $Int, j: $Int :: {$w_div_nat(i, j)} ($unwrap(i)
      >= 0 && $unwrap(j) > 0) ==> (
334       $unwrap($w_div_nat(i, j)) <= $unwrap(i)
335     )
336   }
337
338   // tested
339   axiom $w_div_mul {
340     forall i: $Int, j: $Int :: {$w_div($w_mulI(i, j), j)}
      $unwrap(j) != 0 ==> $w_div($w_mulI(i, j), j) == i
341   }
342
343
344   // tested
345   axiom $w_div_sign {
346     forall i: $Int, j: $Int :: { $w_div(i, j) } $unwrap(j) !=
      0 ==> $sign($unwrap($w_div(i, j))) == $sign($unwrap(i)) *
      $sign($unwrap(j)) || $sign($unwrap($w_div(i, j))) == 0
347   }
348
349   // tested
350   axiom $w_div_mod_mul {
```

```
351      forall i: $Int, j: $Int :: {$w_div(i, j), $w_mod(i, j)}
         $unwrap(j) != 0 ==> ($unwrap(i) == $unwrap($w_mulI(j,
         $w_div(i, j))) + $unwrap($w_mod(i, j)))
352    }
353  }
```

Appendix C

Inter Contract Increase

```
#
# Copyright (c) 2020 ETH Zurich
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.
#

from . import simple_increase

#@ config: trust_casts

token_A: simple_increase
token_B: simple_increase
_diff: uint256
_lock: bool
_init: bool

#@ preserves:
    #@ always ensures: old(self._lock) == self._lock
    #@ always ensures: old(self._init) == self._init
    #@ always ensures: self._lock ==> mapping(self.token_A)[self] == old(mapping(self.token_A)[self])
    #@ always ensures: self._lock ==> mapping(self.token_B)[self] == old(mapping(self.token_B)[self])

# These variables are constant
#@ invariant: self.token_A == old(self.token_A)
#@ invariant: self.token_B == old(self.token_B)

# Once initialized _diff does not change any more
#@ invariant: old(self._init) ==> self._init and self._diff == old(self._diff)

# Invariant we want to prove
#@ inter contract invariant: not self._lock and self._init ==> self._diff == mapping(self.token_A)[self] - mapping(self.token_B)[self]

```

```
34
35 @public
36 def __init__(token_A_address: address , token_B_address:
       address):
37     self._lock = True
38     self.token_A = simple_increase(token_A_address)
39     self.token_B = simple_increase(token_B_address)
40     value_A: uint256 = self.token_A.get()
41     value_B: uint256 = self.token_B.get()
42     assert(value_A >= value_B)
43     self._diff = value_A - value_B
44     self._lock = False
45     self._init = True
46
47
48 @public
49 def increase() -> bool:
50     assert not self._lock
51     assert self._init
52     result: bool = False
53     if self.token_A.get() != MAX_UINT256 and self.token_B.get
       () != MAX_UINT256:
54         self._lock = True
55         self.token_A.increase()
56         self.token_B.increase()
57         self._lock = False
58         result = True
59     return result
```

Appendix D

Contracts Used for Evaluation

D.1 Uniswap default method

```
# @title Uniswap Exchange Interface V1
# @notice Source code found at https://github.com/uniswap
# @notice Use at your own risk

#@ config: trust_casts, no_overflows

from vyper.interfaces import ERC20

token: ERC20

#@pure
@private
@constant
def getInputPrice(input_amount: uint256, input_reserve: uint256, output_reserve: uint256) -> uint256:
    assert input_reserve > 0 and output_reserve > 0
    input_amount_with_fee: uint256 = input_amount * 997
    numerator: uint256 = input_amount_with_fee * output_reserve
    denominator: uint256 = (input_reserve * 1000) + input_amount_with_fee
    return numerator / denominator

#@ ensures: (eth_sold > 0 and self.balance > eth_sold ==> forall({token_reserve: uint256}, token_reserve > 0 ==> result(self.getInputPrice(eth_sold, self.balance - eth_sold, token_reserve)) >= result(self.getInputPrice(eth_sold, self.balance, token_reserve)))) and self.balance == old(self.balance)
@private
def ethToTokenInput(eth_sold: uint256(wei), min_tokens: uint256, deadline: timestamp, buyer: address, recipient: address) -> uint256:
    assert deadline >= block.timestamp and (eth_sold > 0 and min_tokens > 0)
    token_reserve: uint256 = self.token.balanceOf(self)
    tokens_bought: uint256 = self.getInputPrice(as_unitless_number(eth_sold), as_unitless_number(self.
```

```
    balance - eth_sold), token_reserve)
27      assert tokens_bought >= min_tokens
28      assert_modifiable(self.token.transfer(recipient,
    tokens_bought))
29      return tokens_bought
30
31  #@ ensures: msg.value > 0 and old(self.balance) > 0 ==>
    forall({token_reserve: uint256}, token_reserve > 0 ==> \
32      #@ old(result(self.getInputPrice(msg.value, self.balance,
        token_reserve))) \
33      #@ >= result(self.getInputPrice(msg.value, self.balance,
    token_reserve)))
34  @public
35  @payable
36  def f1():
37      self.ethToTokenInput(msg.value, 1, block.timestamp, msg.
    sender, msg.sender)
```

D.2 Modifying Struct in a Loop

```
1   #
2   # Copyright (c) 2019 ETH Zurich
3   # This Source Code Form is subject to the terms of the
        Mozilla Public
4   # License, v. 2.0. If a copy of the MPL was not distributed
        with this
5   # file, You can obtain one at http://mozilla.org/MPL/2.0/.
6   #
7
8
9   CC: constant(int128) = 64
10
11
12  struct Large:
13      a00: int128
14      a01: int128
15      a02: int128
16      a03: int128
17      a04: int128
18      a05: int128
19
20
21  #@ ensures: implies(success(), result().a00 == l.a00 + CC)
22  #@ ensures: implies(success(), result().a00 == result().a01 -
        result().a02)
23  #@ ensures: implies(success(), implies(l.a05 > 0, result().
    a05 == CC * l.a05 + CC))
24  @public
25  def test0(l: Large) -> Large:
26      new_l: Large = l
27      new_l.a05 = 0
28      for i in range(CC):
29          #@ invariant: new_l.a00 == l.a00 + loop_iteration(i)
30          #@ invariant: new_l.a01 == (new_l.a00 + new_l.a02 if
    loop_iteration(i) > 0 else l.a01)
31          #@ invariant: new_l.a02 == l.a02
32          #@ invariant: new_l.a03 == l.a03
```

```
33          #@ invariant: new_l.a04 == l.a04 + loop_iteration(i)
    * (l.a02 + l.a00 + 3) + sum(previous(i)) + i
34          #@ invariant: new_l.a05 == (loop_iteration(i) * l.a05
     + loop_iteration(i) if l.a05 > 0 else 0)
35          new_l.a00 = new_l.a00 + 1
36          new_l.a01 = new_l.a00 + new_l.a02
37          new_l.a04 = new_l.a04 + new_l.a02 + new_l.a00 + 3
38          if l.a05 > 0:
39              new_l.a05 = new_l.a05 + l.a05 + 1
40      return new_l
```

D.3 Altered Version of Stableswap's get_y

```
1  N_COINS: constant(int128) = 3
2
3  #@ ensures: success() ==> result() <= start
4  @public
5  @constant
6  def get_y(start: uint256, b: uint256, c: uint256) -> uint256:
7      assert start >= 2 * c
8      assert b > 0
9      assert c > 0
10     y_prev: uint256 = start
11     y: uint256 = start
12     for _i in range(255):
13         #@ invariant: y <= y_prev and y_prev <= old(y)
14         #@ invariant: y >= c
15         y_prev = y
16         y = (y * y + c) / (2 * y + b)
17
18
19         #@ lemma_assert y_prev * y_prev + c <= y_prev *
    y_prev + 1 * y_prev
20         #@ lemma_assert (y_prev * y_prev + 1 * y_prev) / (2 *
     y_prev + b) <= (y_prev * y_prev + 1 * y_prev) / (2 *
    y_prev)
21         #@ lemma_assert (y_prev * y_prev + 1 * y_prev) / (2 *
     y_prev + b) <= y_prev * (2 * y_prev) / (2 * y_prev)
22
23         # Equality with the precision of 1
24         if y > y_prev:
25             if y - y_prev <= 1:
26                 break
27         else:
28             if y_prev - y <= 1:
29                 break
30         if y < c:
31             break
32     return y
```

D.4 Contract with Cubic Polynomial That Does Not Use Interpreted Lemmas

```
#
# Copyright (c) 2020 ETH Zurich
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.
#

#@ requires: b == c + d
#@ lemma_def distributive(a: uint256, b: uint256, c: uint256, d: uint256):
    #@ a * b == a * c + a * d

#@ lemma_def mul_recursive_step(i: uint256, j: uint256):
    #@ j > 0 ==> lemma.distributive(i, j, j - 1, 1)
    #@ j < MAX_UINT256 ==> lemma.distributive(i, j + 1, j, 1)
    #@ i > 0 ==> lemma.distributive(j, i, i - 1, 1)
    #@ i < MAX_UINT256 ==> lemma.distributive(j, i + 1, i, 1)

#@ lemma_def times_400(i: uint256):
    #@ lemma.distributive(i,   2,   1,   1)
    #@ lemma.distributive(i,   4,   2,   2)
    #@ lemma.distributive(i,   8,   4,   4)
    #@ lemma.distributive(i,  16,   8,   8)
    #@ lemma.distributive(i,  10,   2,   8)
    #@ lemma.distributive(i,  32,  16,  16)
    #@ lemma.distributive(i,  40,   8,  32)
    #@ lemma.distributive(i,  50,  10,  40)
    #@ lemma.distributive(i, 100,  50,  50)
    #@ lemma.distributive(i, 200, 100, 100)


#@ ensures: success() ==> result() <= 63362376
#@ ensures: success() ==> x >= 4
@public
@constant
def get_y(x:uint256) -> uint256:
    assert x <= 400
    #@ lemma_assert lemma.times_400(400) and lemma.times_400(400 * 400)
    #@ lemma_assert 400 * 400 * 400 - 4 * 400 * 400 + 6 * 400 - 24 == 63362376
    #@ lemma_assert (400 - 4) * (400 * 400) + 6 * 400 - 24 == 63362376
    #@ lemma_assert x * x * x - 4 * x * x + 6 * x - 24 == x * (x * x) - 4 * (x * x) + 6 * x - 24
    #@ lemma_assert x * (x * x) - 4 * (x * x) + 6 * x - 24 == (x - 4) * (x * x) + 6 * x - 24
    #@ lemma_assert x >= 4 ==> x * x * x - 4 * x * x + 6 * x - 24 <= 63362376
    #@ lemma_assert lemma.mul_recursive_step(3, 3) and lemma.mul_recursive_step(3, 5) \
        #@ and lemma.mul_recursive_step(3, 7) and lemma.mul_recursive_step(3, 9)
```

```
45      #@ lemma_assert x < 4 ==> x * x * x - 4 * x * x + 6 * x -
        24 < 0
46      return x * x * x - 4 * x * x + 6 * x - 24
```

Bibliography

[1] Brownie: a python-based development and testing framework for smart contracts targeting the ethereum virtual machine. https://github.com/eth-brownie/brownie, 2020. Accessed: 2020-09-09.

[2] Contractguard: a smart contract testing tool that exploits both fuzzing and symbolic execution. https://contract.guardstrike.com/, 2020. Accessed: 2020-09-09.

[3] Ethereum foundation: hive - ethereum end-to-end test harness. https://github.com/ethereum/hive, 2020. Accessed: 2020-09-09.

[4] Ethereum foundation: Pythonic smart contract language for the evm. https://github.com/ethereum/vyper, 2020. Accessed: 2020-09-09.

[5] Ethereum foundation: Smart contract specification language. https://github.com/ethereum/act, 2020. Accessed: 2020-09-09.

[6] Ethereum foundation: The solidity contract-oriented programming language. https://github.com/ethereum/solidity, 2020. Accessed: 2020-09-09.

[7] Ethereum foundation: Tools for testing ethereum based applications. https://github.com/ethereum/eth-tester, 2020. Accessed: 2020-09-09.

[8] Oyente: An analysis tool for smart contracts. https://github.com/melonproject/oyente, 2020. Accessed: 2020-09-09.

[9] pytest 6.0.1. https://pypi.org/project/pytest/, 2020. Accessed: 2020-09-09.

[10] Securify: Security scanner for ethereum smart contracts. https://securify.chainsecurity.com, 2020. Accessed: 2020-09-09.

[11] Slither, the solidity source analyzer. https://github.com/crytic/slither, 2020. Accessed: 2020-09-09.

[12] Smart contract security service for ethereum. https://mythx.io, 2020. Accessed: 2020-09-09.

[13] Smartcheck - a static analysis tool that detects vulnerabilities and bugs in solidity programs . https://github.com/smartdec/smartcheck, 2020. Accessed: 2020-09-09.

[14] Vertigo: a mutation testing framework designed to work specifically for smart contracts. https://github.com/JoranHonig/vertigo, 2020. Accessed: 2020-09-09.

[15] Verx smart contract verification benchmarks. https://github.com/eth-sri/verx-benchmarks/, 2020. Accessed: 2020-10-06.

[16] Viper tutorial. http://viper.ethz.ch/tutorial/, 2020. Accessed: 2020-09-09.

[17] Vyper documentation. https://vyper.readthedocs.io/en/v0.1.0-beta.17/, 2020. Accessed: 2020-09-08.

[18] Phil Daian. Analysis of the dao exploit. https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/, 2016. Accessed: 2020-09-09.

[19] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[20] Michael Egorov. stableswap. https://github.com/curvefi/curve-contract/blob/master/vyper/stableswap.vy, 2020. Accessed: 2020-17-09.

[21] Ákos Hajdu and Dejan Jovanovic. solc-verify: A modular verifier for solidity smart contracts. *CoRR*, abs/1907.04262, 2019.

[22] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, 2014.

[23] Everett Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, 2018.

[24] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[25] Yuri Matiyasevich. *Hilbert's 10th Problem*. MIT Press, 1993.

[26] Michal Moskal. Programming with triggers. *ACM International Conference Proceeding Series*, 01 2009.

[27] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP*, pages 18–20, 2020.

[28] Robin Sierra. Verification of ethereum smart contracts written in vyper. Master's thesis, 2019.

[29] Alexander J. Summers. Encoding to smt. Lecture notes, 2019.

[30] Alfred Tarski. A decision method for elementary algebra and geometry. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 24–84. 1998.

[31] H. Paul Williams. *Logic and Integer Programming*. Springer, 2009.

[32] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[33] Yi Zhang, Xiaohong Chen, and Deajun Park. *Formal Specification of Constant Product (x*y = k) Market Maker Model and Implementation*. 2018.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Verification of Advanced Properties for Real World Vyper Contracts

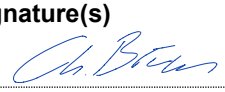
Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Bräm	Christian

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Steinmaur, 17.10.2020	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.