Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cross-Language Verification of Python Modules Written in C

Master Thesis

E. Birling

April 24, 2025

Advisors: Prof. Dr. P. Müller, Dr. M. Eilers

Department of Computer Science, ETH Zürich

Abstract

Modern programs often use multiple programming languages together. For example, Python is a popular language, but many Python applications use parts written in C to be faster or access low-level features. When mixing languages like this, it becomes more difficult to ensure that all parts work together correctly. This thesis looks at how we can prove that Python code using C modules is correct, using deductive verification.

Many tools already exist to verify Python Code or C code individually. Among those, Nagini can verify Python code, and VeriFast can verify C code. But there is no tool that can verify Python code calling C functions. This thesis introduces a new system, called **VFPY**, which helps solve this problem. It allows Python specifications written in Nagini to be translated into VeriFast specifications for C code using Python's C API. With this, users can check that C code used in Python behaves correctly.

We describe how VFPY models Python's behavior inside VeriFast, including important concepts like Python objects, reference counting (for memory management), and exception handling. The thesis also explains how Python specifications can be automatically translated to C and Verifast specifications. We evaluate our system on several real examples, including list handling, attribute access, and using the `gmpy2` math library. These tests show that VFPY works for practical cases and can help ensure Python modules written in C are safe and correct.

Contents

Contents **iii**

1 Introduction **1**

2 Technical Background **3**
2.1 Program Verification 3
2.1.1 Nagini: Verifying Python Code 3
2.1.2 Verifast 4
2.2 Python Modules written in C 5
2.2.1 Python Objects and Python API for C Modules 6
2.2.2 Immutable Values 6
2.2.3 Mutable Aspects 8
2.2.4 Reference Counting 9
2.2.5 Exceptions 13

3 Problem Definition **15**
3.1 Project Specifications 15
3.2 Working Steps 16
3.2.1 Modeling 16
3.2.2 Document the Python API 16
3.2.3 Translation 17
3.2.4 Testing and Verifying 17
3.3 Challenges 18
3.3.1 Duality in Semantics 18
3.3.2 Differences in Verifier Semantics 19
3.4 Focus and Design Choices 21
3.4.1 Supported Specifications 21
3.4.2 Subset of the API Methods To Specify 21
3.4.3 Easy Abstractions vs. Precise Representation 21
3.4.4 Exceptions 22

4 VFPY, Representing the Python Ecosystem using VeriFast **23**
4.1 PyObjects . . . 23
4.2 Immutable Values . . . 23
4.2.1 Representing Python Types . . . 24
4.2.2 Extract: Python Tuples . . . 25
4.3 Reference counting . . . 27
4.3.1 Relative Reference Count . . . 27
4.3.2 Persistently Stored References . . . 29
4.3.3 Reference Count and PyObject Value . . . 30
4.3.4 Taking Advantage of VeriFast's Leak Detection . . . 30
4.4 Attributes . . . 31
4.4.1 Attribute Access Permissions . . . 32
4.4.2 Fractional Permissions . . . 33
4.4.3 Attribute MayCreate Permission . . . 34
4.4.4 Attribute MaySet Permission . . . 35
4.5 IsInstance and IsSubClass . . . 36
4.5.1 Goal . . . 36
4.5.2 Premises . . . 36
4.5.3 Modeling . . . 37
4.6 Lists . . . 39
4.6.1 Content Access Permission . . . 39
4.6.2 Lists and reference counting . . . 40
4.6.3 Iterated Separating Conjunctions . . . 40
4.7 Exceptions . . . 44
4.7.1 Exception as Python Objects . . . 44
4.7.2 Exception Predicate . . . 45
4.7.3 Specifying the API . . . 46

5 Translation **49**
5.1 Facts and Expressions . . . 49
5.1.1 AST Translation . . . 50
5.1.2 VFPY Abstract Syntax Tree . . . 51
5.2 Symbol Translation . . . 53
5.2.1 Context . . . 54
5.2.2 Delayed Access . . . 56
5.2.3 Querying Contexts For Symbols . . . 58
5.2.4 Example: Context Initialization . . . 59
5.3 Translating Specifications . . . 59
5.3.1 Setup . . . 60
5.3.2 PyExceptions . . . 61
5.3.3 Specification Body . . . 61
5.4 Environment Translation . . . 64
5.4.1 Class System . . . 64
5.4.2 Pure Functions . . . 65

5.4.3 Predicates 68

6 Evaluation **69**
6.1 Unit Tests 69
6.1.1 Oracle 70
6.1.2 Practical aspects 70
6.2 Case Study: List Item Extraction 71
6.2.1 Python Specifications 71
6.2.2 Verifying the Method 72
6.3 Case Study: Attribute access 73
6.3.1 Nagini Specifications 73
6.3.2 Verifying the Method 73
6.4 Case Study: GMPY2's Binomial Coefficient Function 74
6.4.1 Python Specifications 74
6.4.2 Translating Specifications into VFPY 76
6.4.3 Method Implementation 77
6.4.4 MPZ Specifications 80
6.4.5 Verification 81

7 Future Works **85**

8 Conclusion **87**

A Annotations for Case Study "List Extraction" **89**
A.1 Automatically Translated VFPY Specifications 89

B Annotations - "Attribute Extraction" Case Study **91**
B.1 Automatically Translated VFPY Specifications 91
B.2 Loop Invariants 92

C Code Repository **93**

Bibliography **95**

Chapter 1

Introduction

As computer programs evolve into increasingly complex systems, testing the code, even thoroughly, does not always suffice to ensure its correctness. Program Verification techniques, allowing to model specifications for a program and prove with absolute certainty that the program implements these, become progressively accessible to more and more languages.

One of them, *Python*, is notoriously known for its flexible behaviors and various possibilities, thereby creating challenging circumstances to reason about: the more complex the possibilities in Python become, the more use cases the verification system must be able to handle to correctly verify arbitrary code. A typical such exotic case is the possibility of embedding *Python Modules written in C* in Python code, thereby allowing the use of sophisticated high-performance C code by Python through a simple wrapper.

Nagini [3] is a modular automated verifier for Python programs written in a statically typed subset of Python. In this context, *modularity* denotes the principle that allows each part of a program to be verified in isolation, under assumptions about its interactions with other parts. This allows the user to verify programs calling functions whose implementation is unknown or ill-suited for automated verification in Python (for example, Python module written in C).

VeriFast, on the other hand, [4] is a modular automated verifier for C and Java programs, supporting a subset of both of these languages.

Program verification is already achievable in both C and Python thanks to Nagini and Verifast; however, nothing exists yet to combine both into a system verifying a Python client of C-implemented modules.

This work focuses on bridging that gap, by providing a *Verifast* environment allowing to express specifications of the Python C API's methods [5] and those of the C module's methods, and by implementing an automated translation

process rephrasing Python *Nagini*-styled annotations of the C module's methods into semantically equivalent *Verifast* annotations.

Chapter 2

Technical Background

This chapter introduces the tools, languages, and verification concepts that form the foundation of this work. We provide an overview of the Python and C verification frameworks involved, and explain key notions such as separation logic and formal specifications.

2.1 Program Verification

Program verification is a range of techniques used to ensure that software behaves as expected by mathematically proving its correctness against a formal specification. By rigorously analyzing code, formal verification allows to identify potential errors or vulnerabilities early, increasing the reliability and security of the verified system.

2.1.1 Nagini: Verifying Python Code

Nagini [3] is a modular automated verifier for Python programs written in a statically-typed subset of Python as defined in [8].

In this subset, all parameters, local variables, and object fields must be annotated with a fixed, static type, which cannot change throughout the execution of the program. Subtyping is supported, allowing objects of a subclass to be used where a superclass is expected. Generic types, such as `List[int]` or `Dict[str, float]`, are also supported.

Implicit Dynamic Frames

Accesses to shared mutable values (e.g. objects passed in reference semantics or global variables) are handled through *Implicit Dynamic Frames* [7], a system of access permissions allowing one to track which heap sections are potentially modified by a callee. This technique is derived from *separation*

logic [6], a formalism considering the heap as a set of disjoint areas, which can thus be reasoned about separately.

2.1.2 Verifast

VeriFast [4] is a modular automated verifier for C and Java programs, using a subset of these languages.

Separation Logic

Separation logic is a formal methods technique used to reason about programs manipulating persistent memory locations. It extends classical Hoare logic by introducing the concept of *separating conjunction*, written as $P * Q$, which means that P and Q hold for disjoint parts of memory [6].

This allows one to describe how different parts of a program operate independently on different memory regions, which is especially useful for reasoning about pointers, heap structures, and concurrent code. VeriFast is built on separation logic and uses it to verify memory safety and correctness properties in C and Java programs, by explicitly tracking which parts of memory a function may access or modify.

Memory Leak Detection

VeriFast detects memory leaks by treating predicate instances as logical memory. If a predicate instance is not correctly released or transferred by the time a function returns, VeriFast will report a leak. This helps ensure memory safety, especially when verifying C modules that interact with Python's reference-counting system.

Lemmas

In VeriFast, a *lemma* is a special kind of function used only during verification.

A lemma is declared and defined like a normal C function marked with annotations: it must have a precondition and a postcondition, and VeriFast checks that the lemma itself is valid. However, since it does not run at program runtime but only helps prove that the code's correctness, only appears within program annotations.

Lemmas are useful when we want to prove the same kind of fact in different places, such as showing that two permissions can be combined or that a list has certain properties. Once a lemma is proved, we can use it like a small helper proof wherever its precondition holds. This makes complex proofs easier to manage and reuse.

Like a C function, a lemma may also be declared without any statement about its definition, case in which its specifications will simply be assumed.

Bigstar

In VeriFast, the `bigstar` construct is used to represent an iterated separating conjunction. It allows one to specify that a certain predicate holds for each element in a domain and that the associated memory regions are disjoint. It is defined as follows in the VeriFast's built-in ghost header files.

```
/*@
predicate bigstar<T>(predicate(T) p, list<T> used);

lemma void bigstar_extract<T>(predicate(T) p, T value);
  requires bigstar<T>(p, ?used) &*& !mem(value, used);
  ensures bigstar<T>(p, cons(value, used)) &*& p(value);

lemma void bigstar_inject<T>(predicate(T) p, T value);
  requires bigstar<T>(p, ?used) &*& true==mem(value, used)
    &*& p(value);
  ensures bigstar<T>(p, remove(value, used));
@*/
```

The bigstar predicate works in trio with the two lemmas declared below.

- The first lemma allows extracting a predicate instance from the `bigstar` predicate instance: a new instance of the argument predicate `p` for the provided value is produced in addition to the `bigstar` instance, but the `bigstar` instance uses its second parameter (`used`, a list of values) to track the values for which the instance of `p` was extracted. It will prevent extracting again an instance of `p` for all values present in the list `used`.
- The second lemma allows for the exact opposite: once extracted for a value `v`, an instance of the predicate `p` can be "inserted back" into the `bigstar` instance. This will remove the predicate instance `p(v)`, but will also remove the value `v` from the list of values for which an instance of `p` has already been extracted. This will allow to extract again an instance of `p` vor `v` in the future.

2.2 Python Modules written in C

To achieve better performance through lower-level instructions or to benefit from greater flexibility by accessing system-level functionalities, modules can be implemented in C and incorporated into Python code. [5]

A specific API provides access for C clients to most of the Python runtime system through a set of methods, macros, and variables, allowing to interact with it from C code (extracting parameters, call Python-declared functions). The C code has an obligation to keep the Python runtime state consistent

under various aspects (reference counting, exception handling or propagation, etc.).

All of that API is included using the following headers.

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
```

2.2.1 Python Objects and Python API for C Modules

While Nagini only focuses on a statically typed subset of Python, the Python environment for C modules has been designed for the most general variant of the language, thus offering all the required support for Python's notorious dynamic typing, i.e. the type of a field, argument, collection item, etc. is not statically determined.

In order to implement that support, the Python C environment represents every Python value as a `PythonObject`.

What this type actually denotes in C is not central: It is declared as `struct` that the user is never supposed to manipulate as such. Rather, the user should use the Python API, which allows safe handling of Python objects, granting consistency in the Python ecosystem for each operation processed.

Consequently, a C module's method visible from Python typically returns and takes `PyObject *` (pointers to Python objects) as arguments.

Example 2.1 *The C declaration of a method in a Python module that could be called from Python code looks as follows:*

```
    static PyObject * some_method(
            PyObject *self,
            PyObject *args);
```

Note that this signature would be used no matter which number of parameters the function takes. All arguments are passed together compressed in a Python tuple object, pointed to by `args`*. This tuple object can be decompressed using the Python API.*

2.2.2 Immutable Values

Any value belonging to a primitive data type (integer, float, boolean) is stored in an *immutable* object in Python. Actually, whenever a variable, object attribute, or list location is described in a Python perspective, it holds a pointer to an immutable Python object when considered from a C module.

Example 2.2 *A simple update of a variable will cause the pointer to a PyObject containing the initial value to be replaced by a reference to another PyObject containing the new value:*

```
# variable x "holds" nothing
x = 1026
# PyObject A containing the value 1026 is created
# variable x "holds" (points to) PyObject A
y = x
# y and x "hold" the same PyObject
x = x + 1
# a PyObject B containing the value 1027 is created
# variable x now "holds" B
print(x is y) #False
# x and y do not "hold" the same PyObject anymore
```

In order to interact with primitive values stored in Python object, one can extract them using the Python API. Similarly, creating new objects from C primitive values is also possible through the API.

Example 2.3 *Increasing the integer stored in a Python integer object (PyLong) can be done as follows. Note that the returned pointer is distinct from the pointer passed as argument.*

```
PyObject* inc(PyObject *self, PyObject * args){
    long x = PyLong_AsLong(PyTuple_GetItem(args, 0));
    PyObject *res = PyLong_FromLong(x+1);
    return res;
}
```

In addition to values from native types, a few composite types allow only immutable values. Let us mention some of them here.

- Tuples - one cannot update an item of a tuple in Python[1]
- Strings - one cannot update the content of a string. For example, the following code will fail.

```
s = "pirling"
s[0] = "b"
```

In fact, *every* PyObject has an immutable part. For instance, even a PyObject with mutable elements (attribute, list content, etc.) is never supposed to change its type: while it is technically possible to manually update a PyObject's type in C code, it is considered extremely unsafe and is not recommended. No matter what it contains, a list should remain a list; no matter what values its attributes take, an instance of a class X should remain an instance of a class X.

[1]This is only true in Python semantics, but some possibilities to bypass this constraint exist. One of them is the possibility of updating tuple items from a C module through a specific method (PyTuple_SetItem) implemented to this end.

But what can be updated in order to interact with the Python world, then?

2.2.3 Mutable Aspects

While objects of primitive types are immutable, more elaborate objects have mutable elements. A simple rule regulates their updates: any mutable location in a Python object will only ever hold PyObjects pointers.

Attributes

An object's attribute is a concrete example of this proposition: When a class instance `z`'s attribute `attr1` is set or updated, what is actually affected is the relationship between `z` and the object pointed to by `z.attr1`. Formally speaking, no value of primitive type is stored in `z.attr1`, but only a pointer to a Python object, itself holding such a value.

Example 2.4 *While* `z.attr1` *gets twice the same value assigned, the objects pointed to are not the same in both cases (a new Python object is created to wrap the value -10 when executing line 8).*

```
1  class A:
2      def __init__(self, x):
3          self.attr1 = x
4  y = -10
5  z = A(y)
6  print(z.attr1 is y) #True
7  print(z.attr1 == y) #True
8  z.attr1 = y+1-1
9  print(z.attr1 is y) #False
10 print(z.attr1 == y) #True
```

Example 2.5 *The Python API methods that update an object's attribute do not return or take as argument values of the attribute's type but only PyObject references.*

```
int PyObject_SetAttr(PyObject *o,
            PyObject *attr_name,
            PyObject *v);
PyObject *PyObject_GetAttr(PyObject *o,
            PyObject *attr_name);
```

List Content

Similarly, a Python list containing integer values does not contain primitive integers in its C representation, but a list of pointers to PyLong objects.

This system allows one list to contain the same pointer several times, which would not be possible under value-only semantics. It also allows a list to contain instances of different Python types.

2.2.4 Reference Counting

While it may be transparent to the user, Python allocates a new PyObject for every value that occurs in the code. Operation `x = x + 1` from Example 2.2, for example, may implicitly allocate a new object, without any explicit call from the user.

However, Python never requires the user to manually de-allocate objects. So how does Python handle the tons of obsolete objects allocated all over a program execution?

Python relies on a reference-counting-based garbage collector to automatically de-allocate unreachable objects: every object X maintains the count of all references pointing to itself that are reachable from the user's perspective.

Consequently, every time a reference to X is stored in a persistent memory location, Python ensures consistency and increases X's maintained reference count. On the other hand, every time a reference to X stored in a persistent location is replaced (e.g. in a field update) or destroyed (e.g. if the object holding the reference gets garbage-collected), X's reference count will be decreased to maintain consistency.

If the reference count of X ever becomes 0, this will trigger the destruction of X and will allow the corresponding memory chunk to be reused to allocate other objects[2].

Consequences of Inconsistent Reference Count

Assume a reference R_X to a PyObject X. If R_x stored in a persistent location without being taken into account in the reference count of X, then, this reference is, in principle, *not safe* to use.

Indeed, if all other references are replaced or destroyed and X's reference count is decremented for each of them, X's reference count will reach zero, and R_X will be invalidated: X will then be garbage-collected and its memory fragment could then be used to allocate another object Y, potentially of another type, thus causing R_x to point to Y.

[2]The acute reader may have noticed that reference counting alone is not sufficient to determine whether a value can still be used. A group of objects that cyclically reference each other may become unreachable by the user, even though none of them ever reaches a reference count of zero. Python addresses this limitation with a separate cycle detector, which operates automatically. Developers of Python modules are not expected to interact with or manage this mechanism

- This could cause a fault if the Y's type differs from X's, implying that Y does not fit the context in which X was expected: for instance, for X a Python long and Y a string, indexing a list using R_x.
- But even worse, this phenomenon could cause an inconsistent behavior to silently occur if Y randomly appears to be a Python Long containing a value different from X's one. This type of bug is particularly insidious, as it can be extremely difficult to detect and reproduce.

Example 2.6 *This example illustrates how easily one can break an the reference count's consistency of an object.*

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>

PyObject* the_object=NULL;
static PyObject* Set(PyObject *self, PyObject *args){
    //capture argument at position 0
    the_object = PyTuple_GetItem(args, 0);
    return Py_None;
    //refcount of *the_object is now inconsistent
}
static PyObject* Get(PyObject *self, PyObject *args){
    return the_object;
    //the_object could now have been invalidated
}
```

Maintaining Consistent Reference Counting in Python Modules

Methods in the Python API's that update a persistently stored reference in a location reachable from Python code are granted to preserve consistency in the Python ecosystem[3].

However, when persistently storing a reference to a Python object in a location unreachable from Python code (for instance, a persistent storage declared in a C module only), and thereby not using the Python API, no mechanism automatically accounts for the stored reference in the object's reference count.

Instead, the Python API offers the user to manually update the reference count of an Object in such circumstances:

```
//increments the reference count
void Py_INCREF(PyObject* o);

//decrements the reference count
```

[3]See some exceptions in subsection2.2.4

```
5 void Py_DECREF(PyObject* o);
6
7 //manually sets the reference count
8 void Py_SET_REFCNT(PyObject *o, Py_ssize_t cnt);
```

Further variants exist as syntactic sugar or to support more elaborate update constraints.

Owned References

A key takeaway of example 2.2.4 is that, in general, one should only access Python objects through pointers that are properly accounted for in the reference count. Such references are said to be *owned*, they are guaranteed to not be invalidated.

Example 2.7 *An owned reference is returned by any method creating a new Python object from scratch. For example, when creating a PyLong object from a C long integer with the following method*

```
1 PyObject *PyLong_FromLong(long v);
```

Borrowed References

In some cases, however, the Python API may return a copy of a reference that has not been accounted for in the reference count. The object from which the reference was retrieved still has the reference, thus the borrowed reference is guaranteed not to be invalidated as long as its *owner* object keeps the ownership on it (by staying alive and not having the owned reference updated).

Such references are called *borrowed* references. Borrowed references are in general unsafe to use for accessing the object's content, but they may be safe to use in a context about which the user knows that the borrowed reference cannot be invalidated (for instance, in a single-threaded setup where no interaction can occur). In addition, borrowed references can always be safely used for pointer comparison.

Example 2.8 *Retrieving an item from a Python list must be done with care: indeed, the following method returns a borrowed reference, not an owned one.*

```
1 PyObject *PyList_GetItem(PyObject *list,
2         Py_ssize_t index);
```

Only references stored in the list remain owned; the returned reference is borrowed. This can be a difficulty in a concurrent setup, as one must ensure that the borrowed reference cannot be invalidated between the retrieval and the time it gets INCREF'd.

There are ways to bypass this limitation, for instance by calling the following method granting ownership on the returned reference.

```
PyObject *PySequence_GetItem(PyObject *list,
                             Py_ssize_t index);
```

Stolen References

Finally, some references owned by the user are considered to be *stolen*: this consists in storing an *owned* reference into another persistent location *without* increasing the reference count. The previously owned reference is then considered borrowed, while the newly stored reference is considered to be owned.

Example 2.9 *The archetypal example of reference theft is the following method, stealing the ownership of the argument* `item`.

```
int PyList_SetItem(PyObject *list,
                   Py_ssize_t index,
                   PyObject *item);
```

Immortal Objects: Preallocated Values

In an optimization perspective, some high-use values are allocated once and for all. Among those are

- integers (Python Longs) in the range `[-5, 256]`
- the unique `None` and the booleans `False`, `True`

Whenever two Python longs in the range of immortal objects appear to be equal, they are also identical. This avoids redundantly allocating many objects to wrap the exact values.

Example 2.10 *Any integer operation returning a value in the range of immortal integers will not result in a new object allocated, but yield the corresponding immortal value instead.*

```
y=1
x=2
print(x-y == y) #True
print(x-y is y) #True
```

```
y=-6
x=-12
print(x-y == y) #True
print(x-y is y) #False
```

Trying to update the reference count of immortal objects will have no effect.

2.2.5 Exceptions

Like every other type or entity, Python exceptions are represented as Python objects. Their behavior obeys the following rules:

- methods of the API may raise exceptions whenever something goes wrong;
- when a Python exception is raised, nothing happens;
- Python expects a C module method to return NULL if an exception is raised at return time, and a PyObject reference otherwise;
- only one exception can be raised at a time (the latest exception raised overwrites the previous one);
- the user may manually raise exceptions, clear exceptions or lookup the raised exception using methods of the API. Some of the most used methods:

```
1  //raise an exception
2  void PyErr_SetString(PyObject *type,
3              const char *message);
4  void PyErr_SetObject(PyObject *type,
5              PyObject *value);
6
7  //clear raised exception
8  void PyErr_Clear();
9
10 //retrieve type of raised exception
11 PyObject *PyErr_Occurred();
12
```

Chapter 3

Problem Definition

3.1 Project Specifications

The task of this work is to implement a technique as a software component involving Nagini and a C verifier (VeriFast), with the following specifications:

1. For a given Python module M written in C, the user provides the system with a Python stub file containing Nagini-based specifications (symbol N) of every function F in M (that is, a required precondition PRE_F^N and a guaranteed postcondition $POST_F^N$).

2. The system then translates PRE_F^N and $POST_F^N$ into VeriFast-based specifications (symbol VF) PRE_F^{VF} and $POST_F^{VF}$ with equivalent semantics.

   It also generates additional specifications $PRE_{py(F)}^{VF}$ and $POST_{py(F)}^{VF}$ to ensure the correctness of the C-written Python module w.r.t. other properties required from such modules (see Sect. **??**).

   In order for the user to interact with the Python world from C extensions, it will be part of this work to produce formal specifications of the Python API methods. The specifications $PRE_{py(F)}^{VF}$ and $POST_{py(F)}^{VF}$ must not only faithfully reflect Nagini specifications (PRE_F^N and $POST_F^N$), but also be compatible with those specifications for methods of the Python API.

3. The user then annotates the module written in C, in order to prove the correctness of following Hoare triple using VeriFast (I_F being the implementation of F):

$$\{PRE_F^{VF} \wedge PRE_{py(F)}^{VF}\} \qquad I_F \qquad \{POST_F^{VF} \wedge POST_{py(F)}^{VF}\} \tag{3.1}$$

3.2 Working Steps

According to the project specifications, at least four great steps in the working process can be isolated.

3.2.1 Modeling

Before translating even the simplest specifications, one needs to establish the concepts on top of which one can translate Nagini specifications into equivalent ones in VeriFast.

Indeed, natively available concepts in C and VeriFast do not always allow us to phrase Nagini specifications.

Example 3.1 *Consider a Python method with following very simple specifications.*

```
def inc(i:int) ->int:
    Requires(i < 1000 and i is not 3)
    Ensures(Result() < 1001 and Result() is not i)
    return i+1
```

```
PyObject* inc_(PyObject* self, PyObject* args);
```

The natively built-in VeriFast and C setup is insufficient to state anything about `i`*:*

- `i`*'s pointer* $P_{\texttt{i}}$ *cannot be used in the specifications, as it is contained the Python Tuple pointed to by* `args`*.*
- `i`*'s value cannot be used either, as it is wrapped in the PyObject* $O_{\texttt{i}}$ *pointed to by* $P_{\texttt{i}}$ *and not accessible as part of the values to which specifications can refer (arguments and return value).*

The goal of this step will consist of creating a new environment capable of representing the Python environment on top of C and VeriFast, in order to be able to phrase (even manually) Nagini specifications in VeriFast.

In later steps of the report, we will refer to this modeling as *VFPY* (**V**eri**F**ast **Py**thon).

3.2.2 Document the Python API

Once VFPY offers enough support to represent concepts from the Python World on top of VeriFast, one should already be able to manually write specifications of methods whose functionality is independent of user-declared entities (methods, classes, global variables, etc.), like methods of the API. Relying on their online documentation, one can phrase their specifications in VFPY.

3.2.3 Translation

Building on top of VFPY, we will continue by developing an extension of Nagini that automates translation of Nagini specifications into VFPY-based specifications.

This part of the work will also require us to make design choices. Indeed, while it is technically possible to manually translate most of Nagini specifications into VFPY ones, automating the process quickly becomes extremely complex. Thus, trade-offs are likely to be required between time at our disposal, precision of the representation, and practicality of available abstractions. For instance, some advanced functionalities may have to be set aside to favor more common ones or keep the modeling simple enough to be safely understood.

This increased complexity is not only a matter of time and resources at our disposal to finish the work, but also a matter of reliability: the more complex the translation process becomes, the more error-prone and difficult to understand it becomes.

3.2.4 Testing and Verifying

Once the translation system is implemented, it is important to test that it behaves as expected and produces correct specifications. In this subsection, we describe how unit tests and case studies will used to prove the reliability and usefulness of the system.

Unit Tests

Having a full battery of unit tests is necessary when developing a complex module that supports a wide variety of cases.

Like in frequent cases, the role of our unit tests will not be to demonstrate correctness of the translation process, but only to ensure stability over fixes and new features added in the system (i.e., that evolutions of the translation process do not break previously working functionalities).

Case Studies

The role of case studies, however, is to more concretely prove the ability of this work's product to achieve its goals by confronting it with real-life challenges.

In order to stick to real-life cases, our goal is to find a preexisting Python module M written in C, itself delegating the work to a method whose VeriFast specifications can be assumed in a modular manner.

M will be treated as a black box and the VeriFast specifications (pre- and post-condition PRE_f^{VF} and $POST_f^{VF}$) of the selected functions f of M will be modularly assumed to hold.

The next step then consists of annotating the C wrapper's code in order to prove the correctness of the wrapper itself, i.e. to prove following assertions for every function F of the Python module:

$$\{PRE_F^{VF} * PRE_{py(F)}^{VF}\}$$

wrapper's code before delegated call to f

$$\begin{gathered}\{PRE_f^{VF}\} \\ \text{delegated call to } f \\ \{POST_f^{VF}\}\end{gathered} \tag{3.2}$$

wrapper's code after delegated call to f

$$\{POST_F^{VF} * POST_{py(F)}^{VF}\}$$

The correct translation of the specifications of the entire function F between Nagini and VFPY has to be done automatically by the translation system mentioned in the previous section.

3.3 Challenges

Developing a translation system between Nagini and VeriFast involves not only implementation work but also careful modeling of Python semantics within a VeriFast-based verification framework. This section presents the main challenges faced during this process: how to accurately model Python features such as dynamic typing, reference counting, and exception handling. It also covers the difficulty of bridging the differences between Nagini and Verifast.

These challenges directly influenced the design decisions described in the next section.

3.3.1 Duality in Semantics

One the one hand, while each Python variable holds one single value (a PyObject pointer), that value is often used in different manners depending on the context in which the variable is used. Thus it is useful to think of some

operations on as happening on references and some operations happening on the values contained in the PyObject.

Example 3.2 *PyLongs* `x` *and* `y` *will be used as references in the expression* `x is y`*, and their wrapped value will be extracted and used in the expression* `x + y`*).*

On the other hand, C, as a much lower-level language, is designed such that every value is retrieved explicitly in each expression it may occur in. Thus, to reproduce the Python semantics in VeriFast, one will need to map each reference to the value it represents, in order to be able to refer to either the value or the reference depending on the context.

Translating Nagini specifications into VeriFast specifications therefore requires one to keep track of the context in which a variable name is used in order to translate it to the correct corresponding value.

3.3.2 Differences in Verifier Semantics

Separation Logic and Implicit Dynamic Frames

Nagini's separation logic (recall section 2.1.1) naturally allows expressing that a access permission to a location is owned by using the exact expression referred to by the permission.

Example 3.3 *The following code requires access to an object attribute, and the same syntax can be used the refer to the value of the attribute.*

```
1 def pymethod(a: object) -> None:
2     Requires(Acc(a.some_field) and a.some_field>0)
3     Ensures(Acc(a.some_field))
4     pass
```

Verifast, on the other hand, requires one to specify the value stored in the location in the same construct as the access permission is stated. Because this stored value is not necessarily known, one usually handles this by representing the value with a new existentially quantified symbol.

Example 3.4 *The following code requires access to an attribute, but a different syntax must be used to refer to the attribute's value, by introducing a new value.*

```
1 void cmethod(struct abc *a);
2 //@ requires a->some_field |->?some_field_val &*&
     some_field_val>0;
3 //@ ensures a->some_field |-> _;
```

It seems important to emphasize at this point that the difference between separation logic and implicit dynamic frames is not a difference in the verifiers' respective semantics: Indeed, both allow us to express the access

permission to a location, the difference is only in the way one can refer to the value in that location. This is more a syntactical than a semantical difference.

However, different verifier syntax induce differences in their expressivity and thus in their semantics. The difficulties mentioned below in this section are only corollaries of this key difference between Nagini's and VeriFast's syntax.

Sub-Expressions and Permission Tracking

A consequence of the previous point is the *scope* in which a VeriFast pattern-defined value is usable: While Nagini never defines values in specifications (and thus never has problems of limited scopes), pattern-defined values in VeriFast are more restrictive about the context in which they can be used.

Example 3.5 *Requiring conditional access permissions in Nagini allows using these permissions under equivalent conditions, simply with occurrences syntactically equal to those from access permissions (here* `a.field` *and* `b.field`*).*

```
1 def pymethod2(a: object, b: object, c: bool) -> None:
2     Requires(Acc(a.f) if c else Acc(b.f))
3     Requires(a.f > 0 if c == True else b.f > 0)
4     Ensures(Acc(a.f) if c else Acc(b.f))
5     pass
```

In VeriFast, however, a pattern-defined acquired permission can only be used in the context in which it was defined, there is no way to track whether another scope could correspond to the same context. The following method cannot be verified because VeriFast's type checker rejects it: the values `af_val` *and* `bf_val` *are undefined when they occur for the second time (line 3).*

```
1 void cmethod(struct abc *a, struct abc *b, bool c);
2 /*@requires (c ? a->f |-> ?af_val : b->f |-> ?bf_val)
3     &*& (c ? af_val > 0 : bf_val > 0); @*/
4 //@ ensures true;
```

Iterated Separating Conjunctions

Iterated Separating Conjunctions (ISC) are a way to state that the predicate holds for all elements an iterable range (domain, list, sequence, set, etc.).

Nagini supports ISCs. They can be phrased as a predicate instance within a `Forall`-quantified expression.

Example 3.6 *The following specification would be allowed for a method under Nagini, assuming a predicate* `P(i: int)` *declared earlier.*

```
1 Requires(Forall(int,
2     lambda i: Implies(i>=0 and i<100, P(i))))
```

VeriFast, however, does not support ISCs, so translating Example 3.5 into VFPY cannot be done on top of native and built-in concepts.

3.4 Focus and Design Choices

The main focus of this work is on accurately modeling the interaction between Python and C, particularly in the context of verifying C modules used within Python programs. Rather than concentrating on the syntactic and semantic differences between separation logic (used in VeriFast) and implicit dynamic frames (used in Nagini), our priority is to develop a sound and practical bridge that captures the semantics of Python API usage in C.

By doing so, we aim to enable reliable cross-language verification while abstracting away from lower-level differences in specification frameworks.

The different steps mentioned in section 3.2 overlooked all the obstacles discussed in section 3.3 that we will have to overcome during the work. In some cases, we will face trade-offs and may have to sacrifice some specifications to favor others. For such cases, we enumerate some design principles for our later design choices throughout the work in the next subsections.

3.4.1 Supported Specifications

All over this work, we will assume that the specifications of a method are well defined (i.e., they must be accepted by Nagini) before being translated into VFPY semantics.

3.4.2 Subset of the API Methods To Specify

Considering the extremely broad scope of the Python C API, we prioritized modeling the subset most essential for verifying typical module behavior.

More specifically, we focused on API methods that retrieve Python values as native C types—such as integers, floats, longs, and booleans. These methods are commonly used in performance-critical modules and form the foundation for reasoning about computations involving primitive data. Modeling them first would therefore allow the verification framework to support a wide range of real-world C extensions, while keeping the amount of work manageable.

3.4.3 Easy Abstractions vs. Precise Representation

Our goal is to model the full complexity of Python's state model to the degree it is represented in Nagini, not some simplified version of it, even if this comes at the cost of more verbose specifications (see concepts presented in section 2.2).

The rationale motivating this choice is that if the user decides to write Python modules in C rather than common Python code, he must have a specific interest in some features specific to such modules. Enforcing simplified high-level semantics is likely to remove the interest of modules written in C.

Example 3.7 *As explained in the technical background, Python lists (containing integers, for instance) do not directly contain integers, but contain references to PyObjects wrapping integers instead. A possibility would be to simplify this view and represent lists as directly containing integers instead of PyObject pointers.*

Under our design principles, Python lists will be considered to contain references, as they appear to do in reality, not directly values.

This choice implies, among others, that we represent *all levels of indirection*. No simplifying assumption will be made here.

3.4.4 Exceptions

Python API methods can raise exceptions. Some are deterministic in the input (e.g. `TypeError` or `OverflowError`) and can be predicted and avoided, while some are nondeterministic (e.g. relating to failures of memory allocation, `MemoryError`) and must simply be accounted for when writing code.

When writing specifications of such methods, one could choose to model all these exceptions by phrasing the relationship with the input for deterministic exceptions, and simply keep open the possibility of a non-deterministic exceptions.

For the sake of simplifying specifications as much as possible and improving their readability and practicality, we will follow the second guideline to specify preconditions for all API functions that are sufficient to ensure that no deterministic exceptions happen (and thus always verify that there are no such exceptions), instead of allowing these exceptions to happen and specify the resulting error state as function of the input.

Example 3.8 *For instance, assume that we want to write specifications for the following method (extracting an integer from a PyLong object):*

```
1 long PyLong_AsLong(PyObject *obj);
```

The precondition should require the argument object to be of the correct type, instead of specifying in the postcondition that an argument of the wrong type will raise an exception.

Inaccurate Documentation

Poorly documented parts of the API will only be translated in an approximate best-effort manner. For example, exceptions potentially raised by a method are not precisely documented online.

Chapter 4

VFPY, Representing the Python Ecosystem using VeriFast

This chapter introduces VFPY, a collection of VeriFast inductive types, fixpoint functions, predicates and lemmas designed to model Python semantics in VeriFast. We describe how core Python concepts, such as objects, types, reference counting, attributes, and exceptions, are represented using VeriFast predicates. Through these models, VFPY enables the verification of C modules used in Python, bridging the gap between high-level Python specifications and low-level C code.

In the following sections, many examples will naturally focus on translating Nagini specifications into VFPY equivalents. We stress here that these examples do not require any prior knowledge about the automated translation process described in the next chapter. The concepts developed in this chapter may also be used to manually translate Nagini specifications into VFPY specifications.

4.1 PyObjects

As mentioned in subsection 2.2.1, the user is never supposed to manually update a PyObject, and should instead rely on API method calls to update the content of a PyObject. Consequently, no access permission to a PyObject's memory fragment or any of its fields will be provided to the user. As a result, no PyObject's memory should be directly manipulated.

4.2 Immutable Values

PyObjects have some immutable properties, such as their type and, in some cases, their value. To model this, we assume that every `PyObject` has an associated value capturing these immutable aspects.

We define a custom inductive type, `PyObj_Val`, to represent this information. This type includes variants for the basic Python types and is used in combination with a predicate, `pyobj_hasval`, to associate a `PyObject*` pointer with its corresponding immutable content:

```
/*@
inductive PyObj_Val = PyBool_v(bool) |
    PyLong_v(int) |
    PyFloat_v(v) |
    ... ;
predicate pyobj_hasval(PyObject * p; PyObj_Val v);
@*/
```

The various types of possible Python values supported by `PyObj_Val` will be discussed in this chapter.

An important point to keep in mind about this predicate is that it does not represent an access permission, as opposed to the most common understanding of VeriFast predicates. It should be understood rather as a statement about the immutable part of a Python object.

This approach already solves the problem of not being able to refer in the specifications to values wrapped in PyObject:

Example 4.1 *The following specifications show how one can refer to the immutable part of PyObject values using an instance of the predicate pyobj_hasval.*

```
int squarePyLong_AsInt(PyObject* a);
//@ pyobj_hasval(a, PyLong(?a_val)) &*& a_val*a_val<=
   INT_MAX;
//@ pyobj_hasval(a, PyLong(a_val)) &*& result==a_val*
   a_val;

```

4.2.1 Representing Python Types

Additionally, an analogous inductive type `PyObj_Type` represents only the *type* without including the contained value, with the following fixpoint function ensuring translation.

```
/*@
inductive PyObj_Type = PyBool_t |
    PyLong_t |
    PyFloat_t |
    ... |
    PyType_t |
    ... ;
```

```
fixpoint PyObj_Type pyobj_typeof(PyObj_Val v) {
    //PyObj_Val -> PyObj_Type
    ...
}
@*/
```

This also allows for representing constant PyObjects, with one more case in the `PyObj_Val` inductive type:

```
/*@
inductive PyObj_Val = ... |
    PyType_v(PyObj_Val)|
    ... ;
@*/
```

4.2.2 Extract: Python Tuples

Some composite types, such as tuples, are considered immutable in our model, following the semantics of the statically-typed subset of Python supported by Nagini.

As explained before, modeling this specific container is required in order to model the way in which methods of Python modules receive positional arguments.

Every `PyObject` is modeled as having both an immutable and a mutable part. The immutable part captures information that never changes during the object's lifetime—such as its type, and in the case of tuples, the number and identities of its items. In our model, a Python tuple is considered immutable, which means that the list of items it contains does not change. Since each item in a tuple is itself a `PyObject*`, we include these pointers directly in the tuple's immutable representation.

The corresponding element in the inductive `PyObj_Type` must contain the type of each item in the tuple.

```
/*@
inductive PyObj_Val = ... |
    PyTuple_v(list< pair<PyObject*, PyObj_Type> >) |
    ... ;
inductive PyObj_Type = ... |
    PyTuple_t(list<PyObj_Type>) |
    ... ;
@*/
```

Example 4.2 *As the Python tuple pointed to by the argument* `args` *is considered to be immutable, both the precondition and the postcondition will state the same things about its immutable content.*

```
def method1(i: int , f: float) -> int:
```

Manually translating the signature of the above method yields the following:

```
PyObject* method1(self PyObject*, args PyObject*);
/*@
requires pyobj_hasval(args;
    cons(pair(?i_ptr, PyLong_t)
    cons(pair(?f_ptr, PyFloat_t), nil))) &*&
    pyobj_hasval(i_ptr, PyLong_v(?i_val)) &*&
    pyobj_hasval(f_ptr, PyFloat_v(?f_val));
@*//*@
ensures pyobj_hasval(args;
    cons(pair(i_ptr, PyLong_t)
    cons(pair(f_ptr, PyFloat_t), nil))) &*&
    pyobj_hasval(i_ptr, PyLong_v(i_val)) &*&
    pyobj_hasval(f_ptr, PyFloat_v(f_val)) &*&
    pyobj_hasval(result, PyLong_v(?res_val));
@*/
```

Although the translation process is covered in the next chapter, it is useful to already introduce this constant part of a method's specifications, which we refer to as the setup. It specifies the method's arguments and remains the same regardless of the Nagini specifications on the function's behavior. Note that assertions about the method's result are not part of the set-up.

Example 4.3 *These bits of modeling already suffice to sketch some specifications for some of the simplest API methods.*

- *The role of this function is to create a new Python Long*[1]

```
PyObject *PyLong_FromLong(long x);
//@ requires true;
//@ ensures pyobj_hasval(result, PyLong_v(x));
```

- *On the other hand, this function can be used to retrieve the integer value wrapped in a PyObject of type PyLong.*

```
long PyLong_AsLong(PyObject *p);
```

[1]The user should however note that these specifications are not complete yet, as they completely neglect the possibility for this function to fail and throw an exception. This will be discussed later in the report

```
2 //@ requires pyobj_hasval(p, PyLong_v(?v)) &*& v
    <= LONG_MAX &*& v>=LONG_MIN;
3 /*@ ensures pyobj_hasval(p, PyLong_v(v)) &*&
4     result == v;@*/
```

The above specifications already take into account the design choice to prevent avoidable exceptions, by using the precondition to require the value wrapped in the PyLong object to fit in a long C integer. The possibility of OverFlow exceptions is thereby removed.

4.3 Reference counting

As described earlier in the technical background, Python modules written in C must explicitly manage reference counts to ensure memory safety. In this section, we explain how we model this behavior in VeriFast in order to verify that reference counts are updated correctly throughout the C code.

Our approach of reference counting will assume freedom from race conditions and modularity. The key assumption is that no persistent location that one user/client code can update may be simultaneously read by any other user: either they all have a fractional (read-only) access, or only of them has the full (read and write) access.

Thus, if a function `f` has sufficient permission to update a persistent location `L`, we can also assume that no one will be able to notice an update made on `L` before `f` returns. Thus, if `f` stores a reference to a PyObject `X` into a persistent memory location without updating `X`'s reference count(i.e. causing `X` an inconsistent reference count), no one else than `f` will be able to notice it before `f` returns.

It therefore suffices to specify in `f`'s postcondition that any PyObject pointer newly stored by `f` into a persistent location must have been taken compensated for by a corresponding call to INCREF at some point (any time) during `f`'s execution. However, thanks to modularity, the exact time(s) at which the reference consistency is broken or restored becomes irrelevant, as long as it holds again when the function returns.

In the next four subsections, we enumerate all constraints that our modeling for reference counting must satisfy, and we show how and why the predicate `pyobj_hasval` already defined earlier satisfies each of them.

4.3.1 Relative Reference Count

In order to keep our modeling compatible with concurrency, we want to consider reference counting modularly.

In this modeling, the predicate `pyobj_hasval` represents the exclusive permission to decrease or maintain one unit in the total reference count.

Our representation will therefore model the relative reference count (i.e. the evolution of the absolute reference count since the function execution started). We distinguish two versions of the relative reference count:

- For a PyObject pointed to by pointer `P` and any method `f`, the *current relative reference count* of `P` at any point in the method `f` is the number of predicate instances `pyobj_hasval(P, P__VAL)` owned at that point of the method.
- In addition, still for the same PyObject pointer and method `f`, the *total relative reference count* of `P` since the beginning of `f`'s execution (i.e. the difference between the initial reference count and the final reference count) is the *difference* between
  - the number of predicate instances `pyobj_hasval(P, P__VAL)` required in the precondition,
  - the number of instances of predicate instances `pyobj_hasval(P, P__VAL)` ensured in the postcondition.

We point out that it now makes sense to have several instances of `pyobj_hasval` on the same pointer to a PyObject `X` in an assertion: they each denote one reference that has been accounted for in `X`'s reference count.

Example 4.4 *VFPY specifications for the methods* `Py_DECREF` *and* `Py_INCREF` *look as follows.*

```
1 void Py_DECREF(PyObject *o);
2 /*@ requires pyobj_hasval(o, ?v) &*&
3     pyobj_hasval(o, v);@*/
4 //@ ensures pyobj_hasval(o, v);
5 void Py_INCREF(PyObject *o);
6 //@ requires pyobj_hasval(o, ?v);
7 /*@ ensures pyobj_hasval(o, v) &*&
8     pyobj_hasval(o, v);@*/
```

`Py_INCREF` *is specified so that only an object that is guaranteed to not have been garbage-collected (for which a reference is owned) can be incref'd. If no reference for* `o` *is owned, it could get invalidated by another thread by the time* `Py_INCREF` *is called.*

On the other hand, `Py_DECREF` *is specified in a way that prevents* `o` *from being garbage-collected. If an object gets garbage collected, all objects pointed to by its attributes (or items, in the case of a list or a dictionary) must also have their reference count decremented. However, since we only consider the reference count of* `o`*, there*

is no way to know whether a call to `Py_DECREF` *will trigger garbage collection of* `o`*, and whether the* `pyobj_hasval` *permissions corresponding to the attributes (or items) pointers should be removed.*

4.3.2 Persistently Stored References

As discussed at the beginning of this section, race condition freedom ensured by VeriFast allows us to express reference counting consistency of a method's implementation in a modular manner, as part of its specifications (i.e. using the common pre- and postconditions).

But what makes such modular specifications sound, then? When do they faithfully describe the reality of the Python runtime?

No matter what its value is (except `NULL`), any (persistent) heap location storing a pointer `P` should be accounted for in the reference count of the object pointed to by `P`. The fact that it is persistent induces that the reference will remain after the function returns, which should therefore be taken into account in the specifications.

Whenever a (non-NULL) pointer `P` pointing to an object `X` is stored in a persistent location `L`, `X`'s reference count must be increased, and the corresponding `pyobj_hasval` must be duplicated on `L` accordingly.

Example 4.5 *The following method captures a reference and correspondingly ensures that the global location holding the captured reference is taken into account by the reference count.*

```
1  PyObject *global = NULL;
2  void capture(PyObject *o)
3  /*@ requires pyobj_hasval(o, ?v) &*&
4          global |-> _;@*/
5  /*@ ensures pyobj_hasval(o, v) &*&
6          global |-> ?global_ptr &*&
7          global_ptr == o &*&
8          pyobj_hasval(global_ptr, v);@*/
9  {
10     Py_INCREF(o);
11     global = o;
12 }
```

Note that, while the possibility to store PyObject pointers in global variables can be specified using VFPY, the automated translation process described in the next chapter does not support this yet.

4.3.3 Reference Count and PyObject Value

If a PyObject is not guaranteed to have a positive reference count (since the object could have been garbage-collected and reallocated since then), referring to any part (including the immutable part) of its value must not be possible.

Although `pyobj_hasval` was initially only used to represent the immutable part of a PyObject's value, it is unsound to consider a PyObject's value independently from its reference count.

Indeed, this separation would potentially allow one to assert something about the immutable part of PyObject `X`'s value, while the reference `X` could have been invalidated at the same time (thus making the `pyobj_hasval` assertion unsound, as `X` is would not even guaranteed to further exist, then).

Using this predicate to represent reference count increments *inherently* enforces that the content of the PyObject is still as asserted, since the reference count increment represented by this assertion *prevents* the PyObject from being garbage-collected.

4.3.4 Taking Advantage of VeriFast's Leak Detection

We would like our modeling to prevent VeriFast from verifying the program if the reference count becomes inconsistent in either direction (i.e. one persistently stored reference is not accounted for, or the reference count takes a reference into account that does not exist anymore).

All predicate instances are treated by VeriFast like memory fragments are. This implies the following interesting facts:

- VeriFast will ensure that no `pyobj_hasval` instance is leaked. This faithfully reflects the fact that an increment of the reference count of a PyObject `X` should not be maintained if no persistent location correspondinlgy holds a reference to `X`.

  Example 4.6 *Calling the method* `Py_INCREF` *on a PyObject pointer* `P` *twice when storing* `P` *in only one persistent location will trigger VeriFast's leak detection.*

```
1 PyObject *global = NULL;
2 void capture(PyObject *o)
3 /*@ requires pyobj_hasval(o, ?v) &*&
4         global |-> _;@*/
5 /*@ ensures pyobj_hasval(o, v) &*&
6         global |-> ?global_ptr &*&
7         global_ptr == o &*&
8         pyobj_hasval(global_ptr, v);@*/
```

```
{
    Py_INCREF(o);
    Py_INCREF(o);
    //NOTE: this assertion succeeds
    //@assert pyobj_hasval(global_ptr, v) &*&
   pyobj_hasval(global_ptr, v);
    global = o;
    //NOTE: the method cannot be verified, as one
    instance of pyobj_hasval(global_ptr, v) is
   leaked
}

```

- VeriFast will also ensure that no `pyobj_hasval` instance is missing. This faithfully reflects the fact that if a reference to a PyObject `X` is newly stored in a persistent location, there must be a corresponding increment in the reference count.

  Example 4.7 *If one forgets to call* `Py_INCREF` *on a reference when persistently storing it, that will cause the verification to fail, since the postcondition cannot be satisfied.*

```
PyObject *global = NULL;
void capture(PyObject *o)
/*@ requires pyobj_hasval(o, ?v) &*&
        global |-> _;@*/
/*@ ensures pyobj_hasval(o, v) &*&
        global |-> ?global_ptr &*&
        global_ptr == o &*&
        pyobj_hasval(global_ptr, v);@*/
{
    Py_INCREF(o);
    global = o;
    //NOTE: the method cannot be verified, as one
    instance of pyobj_hasval(global_ptr, v) is
   missing to satisfy the postcondition
}

```

4.4 Attributes

Our representation of access permission to PyObject attributes imitates standard separation-logic points-to predicates in their semantics: those consist of a pointer to the target struct, a field name, and the field value.

```
struct Person {
    int age;
    float height;
};
void inc_age(struct Person* p);
//@ requires p->age |-> ?age_v;
//@ ensures p->age |-> ?age_v2 &*& age_v2 == age_v+1;
```

4.4.1 Attribute Access Permissions

In order to keep our representations consistent with the VeriFasts ones, we try to mimic these semantics by declaring our own predicate analogously:

```
/*@
predicate pyobj_hasattr(PyObject* target,
    list<char> name,
    PyObject* attr_ptr);
@*/
```

In order to represent the value taken by the attribute in value semantics, an additional instance of `pyobj_hasval` must be created to describe the content of `Y` pointed to by `X.a`.

Note that we thereby soundly model the semantics of Python C extensions[2] in which a PyObject `X`'s attribute `a` points to another PyObject`Y`. This allows one to use PyObject attributes in both refernce semantics (comparing the pointers stored as attributes of PyObjects `a` and `b`), or in value semantics (comparing the values wrapped in the PyObjects `c` and `d` pointed to by attributes of PyObjects `a` and `b`).

Example 4.8 *Assertions about an attribute can thus be translated into VFPY in both value and reference semantics as follows.*

```
Acc(x.f) and x.f is not i and x.f == i
```

```
/*@ //assume symbol x_ptr has been introduced earlier
//        Acc(x.f)
pyobj_hasattr(x_ptr, "f", ?x_DOTf_ptr)  &*&
pyobj_hasval(x_DOTf_ptr, ?x_DOTf_val)  &*&
//           x.f is not i
x_DOTf_ptr != i_ptr  &*&
//           x.f == i
x_DOTf_val == i_val;
@*/
```

[2]Explained in subsection 3.4.3

4.4.2 Fractional Permissions

Nagini supports partial ownership using fractional permissions, as in the following example:

```
def merging_(x: someclass):
    Requires(Acc(x.attr, 1/2) and Acc(x.attr, 1/2))
    Ensures(Acc(x.attr))
```

Let us now consider a possible VFPY translation of these specifications.

```
PyObject* merging_(PyObject* self, PyObject* args)
/*@  requires pyobj_hasval(args, PyTuple(
    cons(pair(?x_ptr, someclass), nil)) &*&
    pyobj_hasval(x_ptr, _) &*&
    [1/2]pyobj_hasattr(x_ptr, "attr", ?x_attrptr) &*&
    [X]pyobj_hasval(x_attrptr, ?x_attr_val)
    [1/2]pyobj_hasattr(x_ptr, "attr", x_attrptr) &*&
    [X]pyobj_hasval(x_attrptr, x_attr_val);
    @*/
/*@  ensures pyobj_hasval(args, PyTuple(
    cons(pair(?x_ptr, someclass), nil)) &*&
    pyobj_hasval(x_ptr, _) &*&
    pyobj_hasattr(x_ptr, "attr", x_attr_ptr) &*&
    pyobj_hasval(x_attr_ptr, x_attr_val);
    @*/
{

}
```

How would one the fraction`1/2` be used in the translation? More precisely, should the fractions only apply to `pyobj_hasattr` instances, or also to the instances of `pyobj_hasval` applied to the attribute pointer?

We discuss both possibilities in the following. We assume the above context and will then focus on the attribute access permissions.

- If `X` is chosen as $\frac{1}{1}$. One will have a total of $\frac{1}{1} + \frac{1}{1} = 2$ `pyobj_hasval` instances, one more than specified in the postcondition. This will result in either of the following cases.
  - If no function call is made to consume this additional predicate instance, VeriFast will complain about a leaked permission.

    This modeling does not allow one to verify trivially correct cases; thus, it is *incomplete*.

  - If a function call is made to consume this additional predicate instance, the Python runtime may reach an inconsistent state, and VeriFast will not complain.

    Calling `Py_DECREF`, for instance, will get rid of the redundant predicate instance, but will put the Python runtime in an inconsistent state. Indeed, the reference `x_attr_ptr` persistently stored in `x.f` will not be accounted for in the reference counting anymore. The PyObject pointed to by `x.f` would potentially be garbage collected while `x.f` still pints to it.

    Tjishis modeling additionally allows verifying cases that are trivially incorrect, it is therefore *unsound*.

- If `X` is chosen as $\frac{1}{2}$, the fractions of `pyobj_hasval` instances derived from the precondition sum up to $\frac{1}{1}$, as expected in the postcondition. This fits our needs exactly, as this perfectly matches the above Nagini Specifications asserting the conservation of the Fractional permissions.

This example gives the intuition that fractional access permission on attributes should propagate to the corresponding `pyobj_hasval` instance

Meaning of Fractional pyobj_hasval Permissions

Let us recall from section 4.3 that an instance of `pyobj_hasval` is used to represent the access permission to one incremented unit in the total reference count.

If the access permission to an attribute is fractional, the user loses the ability to update this attribute pointer, but keeps the ability to read it (with the guarantee that no one will update it). As a direct consequence, the user should have the same guarantee that the attribute reference will not get invalidated *but* should not get the permission to invalidate it. One can phrase this in a simpler manner as *the user can update the attribute* $\Leftrightarrow$ *the user entirely owns the attribute reference* Thus, it does make sense to assign the same fraction to the `pyobj_hasval` instance applied to the pointer of an attribute whose access permission is only fractionally owned.

4.4.3 Attribute MayCreate Permission

Nagini's specification language has an assertion MayCreate(o, fieldName) that states that said field does not currently exist, and represents the permission to create it. VFPY represents this permission with a predicate using a similar syntax and identical semantics:

```
1 /*@
2 predicate pyobj_maycreateattr(PyObject* p,
```

```
    list<char> name);
@*/
```

4.4.4 Attribute MaySet Permission

Analogously, Nagini supports a permission to set an attribute on Python objects, either by creating it or by updating it. This time, however, we decided to pick a representation slightly different from Nagini's representation:

```
/*@
predicate pyobj_maysetattr(
    PyObject* i,
    list<char> n,
    option<PyObject*> out) = switch(out) {
        case none: return pyobj_maycreateattr(i, n);
        case some(x): return pyobj_hasattr(i, n, x);
    };
@*/
```

The rationale for this is that the permission `pyobj_maysetattr` is supposed to express that one of both permissions `pyobj_maycreateattr` or `pyobj_hasattr` is owned. Whenever the parameter `out` is `none`, the instance of `pyobj_maycreateattr` may be turned The interesting advantage that we can take of this definition is that if the value parameter `out` is unknown, the user will have to handle both cases if the instance of `pyobj_maysetattr` is opened.

Any method from the Python API setting and reading an object `X`'s attribute `a` will require an instance `pyobj_maysetattr(X, a, _)`.

The user can easily derive this permission from both `pyobj_maycreateattr(x,a)` and `pyobj_hasattr(x,a)` by `@close`-ing `pyobj_maysetattr(X, a, _)` once either of them is owned.

Example 4.9 *The following functions all verify.*

```
void m(PyObject *obj)
//@requires pyobj_maycreateattr(obj, "attr_name");
//@ensures pyobj_maysetattr(obj, "attr_name", none);
{
//@close pyobj_maysetattr(obj, "attr_name", none);
}

void m2(PyObject *obj)
//@requires pyobj_maysetattr(obj, "attr_name", none);
//@ensures pyobj_maycreateattr(obj, "attr_name");
{
```

```
12 //@open pyobj_maysetattr(obj, "attr_name", none);
13 }
14
15 void m3(PyObject *obj)
16 //@requires pyobj_maysetattr(obj, "attr_name", some(?
      v));
17 //@ensures pyobj_hasattr(obj, "attr_name", v);
18 {
19 //@open pyobj_maysetattr(obj, "attr_name", some(v));
20 }
```

4.5 IsInstance and IsSubClass

4.5.1 Goal

This section describes a modeling approach that enables us to write the specifications of a specific method from the Python API:

```
1 int PyObject_IsInstance(PyObject *ins, PyObject *cl);
2 /** Return 1 if ins is an instance of the class cl or
      a subclass of cl, or 0 if not. In error, returns
     -1 and sets an exception. **/
```

An immediate problem arising when facing this task is the difficulty of writing case-independent specifications for a method whose output relies on a non-static factor: the class system declared by the user. Our approach to solving this issue is to statically define a fixpoint function that will determine from values dynamically created to represent the class system and passed as arguments.

4.5.2 Premises

PyObjects Representing Types

The acute reader will deduce from the specifications of this `PyObject_IsInstance` that Python types are also represented by PyObjects. For this reason, we declare an additional value in the inductive type `PyObj_Val` to represent the content of PyObjects that carry a type as value. The type contained by such a PyObject belongs to their immutable part.

```
1 /*@
2     PyObj_Val = ... | PyType_v(PyObj_Type) | ... ;
3 @*/
```

From this point on, `PyObj_Type` is no longer just a helper construct; it now plays a semantic role in the specification logic. For example, it is used to

represent the type contained in a `PyObject`, and to reason about relationships such as subtyping and class membership.

Python Class Instances

Whenever a PyObject appears to be an instance of a non-built-in Python class [3], it will be represented as a new kind of immutable PyObject value, `PyClassInstance_v`, containing enough information to allow one to deterministically state about a pair of Python classes whether one is a subclass of the other. In addition, VFPY's modeling for Python classes could also be used to represent the described class as a type.

```
/*@
    PyObj_Val = ... | PyClassInstance_v(PyClass) |
   ... ;
    PyObj_Type =  ... | PyClass_t(PyClass) | ... ;
@*/
```

4.5.3 Modeling

We represent a class using the following inductive type:

```
/*@
inductive PyClass =
    PyClass(list<char>, PyClass, list<PyObj_Type>) |
    ObjectType;
@*/
```

We briefly summarize the role of each of the arguments in the following.

- The first parameter represents the *name* of the class and is given as a string, natively modeled as a `list<char>` in VeriFast.
- The second parameter denotes the parent Python class. The base case is the value `ObjectType` corresponding to Python's `object` type.
- The last parameter represents type arguments in the case of generic types. Note that a Python class representation should not hold all its type arguments, but only those coming *in addition* to the parameter types of the parent Python class.

Example 4.10 *The Python class hierarchy on the left can be represented with the VFPY system on right.*

[3] Specifically in this context, some built-in and most basic types types are not denoted as Python classes. One should create a class extending these in the Python subset supported by Nagini.

```
class A_:
    pass


class B_(A_):
    pass


class C_(A_):
    pass


class D_(B_):
    pass
```

```
/*@ fixpoint PyClass A_(){
  return PyClass("A", ObjectType,
   nil);
}

fixpoint PyClass B_(){
  return PyClass("B", A_, nil);
}

fixpoint PyClass C_(){
  return PyClass("C", A_, nil);
}

fixpoint PyClass D_(){
  return PyClass("D", B_, nil);
}
@*/
```

Example 4.11 *An instance of class* `D_` *could be represented as follows in some method specifications.*

```
def handle_some_instance(PyObject* p);
//@ requires pyobj_hasval(p, PyClassInstance_v(D_()));
//@ ensures ...;
```

A fixpoint function can now deterministically establish from two values of `PyClass`, `A` and `B`, whether `A` is a subclass of `B`.

```
/*@
fixpoint bool issubclass(PyClass cls1, PyClass cls2)
   {...}
@*/
```

We defined additional functions to generalize this function to handle all other values of type `PyObj_Type`.

```
/*@
fixpoint bool issubtype(PyObj_Type t1, PyObj_Type t2)
   {...}
fixpoint bool isinstance(PyObj_Val v, PyObj_Type t)
   {...}
@*/
```

Beside facilitating the phrasing of specifications of methods like `PyObject_IsInstance` or `PyObject_IsSubclass`, these functions allow to translate in-situ the cor-

responding calls to Nagini pure functions potentially made in a Python method's specifications.

Example 4.12 *Reusing the VFPY class system declared in example 4.10, one could verify following assertions:*

```
//@ assert issubclass(B_, A_)==true;
//@ assert issubclass(A_, A_)==true;
//@ assert issubclass(A_, B_)==false;
//@ assert isinstance(PyClassInstance_v(C_), A_)==
   true;
//@ assert issubtype(PyClass_t(D_), PyClass_t(A_))==
   true;
//@ assert issubtype(PyLong_t, PyClass_t(A_))==false;
```

4.6 Lists

Python has several built-in, mutable container types, such as lists and dictionaries, which are also supported in the statically-typed subset used by Nagini. In this section, we focus on how to represent Python lists within our VeriFast-based model, including how to express their structure and access permissions.

4.6.1 Content Access Permission

Nagini only has *collective* access permission for lists: The `list_pred(l)` permission is required to access *any* item of `l` as a whole.

In addition, similarly to attributes, Python represents list items as references to PyObjects, which means that the access permission of a Python list should actually give access to a sequence of PyObject pointers.

The most intuitive translation of `list_pred` into VFPY is as simple as this:

```
//@ predicate pylist_hasitems(PyObject *, list<
   PyObject *>);
```

Other types of Python containers (sets, dict, etc.) could one day be handled too and would have to benefit from a flexible modeling designed today. We thus decided to use a slightly more complex (but also more flexible) modeling.

```
/*@
    inductive PyObj_Content = List(list<PyObject*>);
    predicate pyobj_hascontent(PyObject *,
   PyObj_Content c);
@*/
```

Having found such a simple representation, let us now examine the various concomitant challenges.

4.6.2 Lists and reference counting

Let us recall that an indexed Python list `l[i]`, like any variable or attribute accessed, may be used in both reference *and* value semantics. Assuming we now have a Python list object whose content is described as follows, how can we represent both cases in VFPY?

```
int listmethod(PyObject * p, int i);
/*@
    requires pyobj_hasval(p, _) &*&
        pyobj_hascontent(List(?ptrlist));
@*/
```

The expression `l[i]` used in reference semantics simply denotes the pointer to the PyObject pointed to by the `i`-th item of `l`. In the above case, it can be accessed using VeriFast native constructs as `nth(i, ptrlist)`.

However, in the case of value semantics, one must have the predicate `pyobj_hasval` on `nth(i, ptrlist)` to access `l[i]` as a value.

In addition, we would like to be able to access the list item in *value* semantics with non-constant indices (like `i`), in the same manner as the expression `nth(i, ptrlist)` supports a non-constant index in *reference* semantics.[4].

Our modeling of reference counting requires that every reference stored as a list item be accounted for in the item's reference count. But how to state that `pyobj_hasval` instances are held for all items of a list?

4.6.3 Iterated Separating Conjunctions

As mentioned in section 3.3.2, VeriFast supports iterated separating conjunctions with the Bigstar concept introduced in section 2.1.2. In order to model Nagini's iterated separated conjunction, we use a construct that imitates the Bigstar one.

```
/*@
list_forallpred<intype, outtype>(
    list< pair<intype, outtype> > l,
    predicate(intype, outtype) p,
    list_forallcond cond,
    list<int> out);@*/
```

The different parameters of this predicate are explained below.

[4]This consistency in support between value and reference semantics with non-constant indices will avoid many complications in the translation process.

Input-Output Pair List

Denoted by `l` in the declaration, this list contains pairs of input and output of the predicate to be called on the whole list. Typically, the inputs consist of PyObject pointers (the PyObject references stored in the list).

We chose to use a list of input-output pairs instead of two separate lists, since the latter does not inherently enforce that both lists have the same length, while a list of pairs can easily be decomposed into two different lists of (acknowledged to have equal lengths).

Example 4.13 *Decomposing a list of pairs into two lists of equal lengths can be done as follows*[5]

```
1 void nothing()
2 //@ requires list_forallpred(?l, _, _, _);
3 //@ ensures list_forallpred(l, _, _, _);
4 {
5   //@assert some(map(fst, l)) == some(?input);
6   //@assert some(map(snd, l)) == some(?output);
7   //@assert length(input) == length(output);
8 }
```

Predicate

This is the predicate for which the `list_forallpred` gives permission. It only takes one input and one output parameter, but this limitation is easy to mitigate by defining predicates to match it, or by using predicate constructors.

List of Extracted Indices

Analogously to the `Bigstar`'s second argument storing elements for which a predicate instance was already extracted (section 2.1.2), `list_forallpred` stores extracted indices in a list held as the last argument, called `out` in the above definition. Whenever an index `i` is stored in this list, it denotes that the predicate instance was already extracted for the `i`-th item and can therefore not be extracted anymore.

Lemmas and Exclusions

Suitably parameterized instances of `list_forallpred` now represent a permission owned on all PyObject pointers in a list. However, they are pointless if the user is not provided with a way to retrieve a permission instance for some concrete index and use it to verify a method.

[5] A trivial autolemma had to be added in VeriFast built-in lemmas with a pull-request to allow to state this.

To achieve this, we have also declared two lemmas analogously to Bigstar's [6] extraction and insertion lemma, somewhat more elaborate, however.Instead of passing the requested value (a PyObject pointer) to extract or insert, the user should call these lemmas with a requested *index*.

```
/*@
lemma void list_forallpred_extract
    <intype, outtype, intermtype>
    (   list<pair<intype, outtype> > L,
        predicate(intype, intermtype) p,
        int idx
    );
requires [?f]list_forallpred(L, p, ?cond, ?out) &*&
    list_forallcondeval(cond, idx) == true &*&
    !mem(idx, out) &*& idx<length(L) &*& 0<=idx;
ensures [f]list_forallpred(L, p,
    cond, cons(idx, out))
    &*& [f]p(fst(nth(idx, L)), snd(nth(idx, L)));
@*/
```

This lemma checks if the requested index satisfies all the required conditions and extracts an instance of `p` for the list `L` and the index `idx` requested by the user. The required conditions on the index `idx` are the following:

- `idx` is in `L`'s bounds;
- `idx` is not among indices already extracted (`out`);
- `idx` satisfies the pre-defined particular condition `cond`.

If the extraction succeeds, the requested index is added to the list of indices already extracted.

Example 4.14 *Let us consider a simple use case of a Python list in Nagini:*

```
def method_(l: List[int], i:int)->None;
    Requires(list_pred(l) and 0 <= i and i < len(l)
   and l[i]==14)
```

The above specifications Nagini using list predicate can be manually translated as follows. One can separate these annotation in several key points:

- *the static VFPY environment providing global fixpoint functions, fixpoint constants and predicates to use in specifications of potentially several methods;*

```
/*@
```

[6] See section 2.1.2

```
2     predicate pyobj_hasPyLongval(PyObject* pyobj,
      int v) = pyobj_hasval(pyobj, PyLong_v(v));
3 @*/
```

- *the setup of the method is omitted for the sake of simplicity, but the reather can simply assume that it defines the symbols ;*
- *the translation of* `list_pred(l)`*, providing access to pointers and values of all list items additionally to the length of the list;*

```
1 /*@
2     &*& pyobj_hascontent(l_ptr, List(?
    l_content_ptr))
3     &*& list_forallpred(?l_content,
    pyobj_hasPyLongval, True, nil)
4     &*& map(fst,l_content) == l_content_ptr
5     &*& some(map(snd, l_content)) == some(?
    l_content_val)
6 @*/
7
```

- *the assertion* `0 <= i`*;*

```
1 /*@
2  &*& 0 <= i_val
3 @*/
```

- *the assertion* `i < len(l)`*;*

```
1 /*@
2  &*& i_val < length(l_content)
3 @*/
```

- *the assertion* `l[i]==14`*.*

```
1 /*@
2  &*& nth(i_val, l_content_val) == 14
3 @*/
```

Conditions

Forall conditions are an additional concept defined as an inductive type and a fixpoint function allowing to express a subset of $\mathbb{N}$

```
1 /*@
2 inductive list_forallcond = True |
3     lt(int a) |
```

```
4     lte(int a) |
5     and(list_forallcond c1, list_forallcond c2) |
6     ... ;
7 fixpoint bool list_forallcondeval(list_forallcond c,
    int x){
8     switch(c){
9         case True: return true;
10        case lt(a): return x<a;
11        case lte(a): return x<=a;
12        case and(c1, c2): return list_forallcondeval(
    c1, x) && list_forallcondeval(c2, x);
13        ...
14    }
15 }
16 @*/
```

Together, they allow selecting a subset of the indices that would naturally be represented by this predicate by default (under condition `True`, corresponding to $\{0, \ldots, length(l)\}$)

4.7 Exceptions

In Python C extensions, exceptions are not thrown in the traditional C sense, but are instead signaled by returning `NULL` and setting a global exception state. This state can then be retrieved using API methods such as `PyErr_Occurred` or `PyErr_Fetch`. As explained in the technical background (Section **??**), only one exception can be active at a time, and the responsibility for managing it lies with the C module developer.

To faithfully verify the behavior of C modules interacting with Python, our model must represent not only when exceptions can occur, but also what exception object is currently raised, and how it is propagated or cleared. This section describes how exceptions are encoded in our VFPY model, and how we specify and verify the behavior of API methods with respect to exception handling.

4.7.1 Exception as Python Objects

Python exceptions are also considered as PyObjects: they can be retrieved in a variable using a try-catch block in Python code; they can be retrieved via methods`PyErr_Fetch` or `PyErr_Occured` in Python Modules written in C.

As an immutable part of every PyObject, the type of exceptions must also be represented in their `pyobj_hasval` instance.

Class System

Since Nagini allows one to extend Python's `BaseException` type, the VFPY exceptions' immutable part must be represented as a `PyClassInstance_v` to let the user declare additional exception types and instantiate them.

The following class system is predefined for all use cases.

```
/*@
fixpoint PyClass BaseException(){
  return PyClass("BaseException", ObjectType, nil);
}
fixpoint PyClass Exception(){
  return PyClass("Exception", BaseException, nil);
}
fixpoint PyClass ValueError(){
  return PyClass("ValueError", Exception, nil);
}
fixpoint PyClass MemoryError(){
  return PyClass("MemoryError", Exception, nil);
}
...
@*/
```

Instantiating Exceptions

An exception can now be simply defined as an instance of the exception types, using the `PyClassInstance_v` value of the inductive type `PyObj_Val`.

4.7.2 Exception Predicate

To faithfully represent the behavior of Python exceptions, a simple predicate with one argument suffices. It represents the Python exception flag according to criteria enumerated in subsection 2.2.5.

```
//@ predicate PyExc_Flag(option<PyObject*>);
```

Semantically, this definition describes that no exception is raised when this instance of this predicates contains a `none` value, while an exception is raised when it contains a `some(?x)` value. In the latter case, an instance of `pyobj_hasval` is also owned to describe the type of the raised exception and reflect the reference count on the PyObject representing the exception.

Consequently, the most typical occurrence of the `PyExc_Flag` will look like this:

```
PyObject* someAPImethod(...);
```

```
/*@
    requires PyExc_Flag(e) &*& switch (e) {
        case none: return t == none;
        case some(x): return t==some(?y) &*&
   pyobj_hasval(x, PyClassInstance_v(y));
    };
@*/
//@ ensures ...
```

This is not really convenient: Apart from being a bit verbose, it prevents us from using the exception type somewhere else in the specification, as the value `x` can only be used in the limited scope of case `some(x)`.

To mitigate this downside, we have defined the following predicate. This variant will be used in the automatic translation process.

```
/*@ predicate PyExc(
    option<PyObject*> e,
    option<PyClass> t) =
        PyExc_Flag(e) &*&
        switch (e) {
            case none: return t == none;
            case some(x): return t==some(?y) &*&
               pyobj_hasval(x, PyClassInstance_v(y));
        }; @*/
```

4.7.3 Specifying the API

API methods have different ways of raising exceptions, among which one can distinguish at least these four.

Differently Triggered

1. Some exceptions are raised deterministically and can be modeled depending on the input.

   Example 4.15 *The following method raises predictable exceptions, depending on its input arguments.*

   ```
   long PyLong_AsLong(PyObject *obj);
   ```

   Some examples include: an OverflowError if the PyObject's value does not fit in a C long, or a TypeError if the object is not an integer or cannot be interpreted as one.

By design choice, as explained in **??**, we will never specify such exceptions as raised in any postcondition. Instead, the precondition will forbid the conditions that would raise such exceptions.

Example 4.16 *The above method could be specified as follows:*

```
long PyLong_AsLong(PyObject *p);
/*@ requires pyobj_hasval(p, PyLong_v(?v))
    &*& v <= LONG_MAX &*& v >= LONG_MIN;@*/
/*@ ensures pyobj_hasval(p, PyLong_v(v))
    &*& result == v;@*/
```

Since this method can raise no (non-deterministic) exception, it does not even require the exception predicate.

2. Some exceptions are raised non-deterministically and cannot be modeled depending on the input.

Example 4.17 *The specifications of the method* `PyLong_FromLong`*, creating a new Python long object from a native C long integer, can be phrased as follows. The nondeterministic exception could be raised by a failed memory allocation of the new Python object, a case in which the method would raise the exception and return* `NULL`*.*

```
PyObject* PyLong_FromLong(long v);
//@ requires PyExc(?exc_old, ?exc_old_cl);
/*@ ensures PyExc(?exc_new, ?exc_new_cl) &*&
    result == NULL ? (
        exc_new != exc_old
        &*& exc_new == some(?x)
        &*& exc_new_cl == MemoryError()
    ):(
        pyobj_hasval(result, PyLong_v(v)) &*&
        exc_new == exc_old &*&
        exc_new_cl == exc_old_cl
    );@*/

```

Return Values and Exceptions

There are different ways in which return values may reflect exceptions.

- In some cases, the return value explicitly reflects that an exception occurred. As an example, consider `PyLong_FromLong` (example 4.17), returning NULL in the case an exception is raised.

- Some exceptions are raised silently or ambiguously: The return value does not reflect the exceptional behavior. As an example, consider `PyLong_AsLong` (example 4.16), returning -1 in case of an exception, while it could legitimately return -1 as a result of the long integer extraction.

Combining both

Both distinctions can be combined, thus forming four types of exception:

- we prevented silent deterministic exceptions with the precondition;
- explicit deterministic exceptions too;
- explicit non-deterministic exceptions are modeled using `result`, the output value;
- silent non-deterministic exceptions can be represented as follows, even using none of the input or output values.

```
1 void nondet_silent_exception(PyObject * a, ...);
2 //@ requires PyExc(?exc_ptr_old, ?exc_type_old)
     &*& ...;
3 /*@ ensures PyExc(?exc_ptr_new, ?exc_type_new)
     &*&
4      (exc_ptr_old!=exc_ptr_new)?
5      ( ... )//behavior in case an exception was
     raised
6      :( ... );//expected behavior
7 @*/
```

Chapter 5

Translation

This chapter describes the automated translation process that takes Python specifications written using Nagini and turns them into equivalent VeriFast specifications that can be verified in the VFPY environment. The translation must bridge both syntactic and semantic gaps between the two systems, while preserving correctness and modular reasoning.

We begin by explaining how the translation process interprets and converts facts and expressions from Nagini's annotated Python code. We then detail how symbols, specifications, and the verification environment are mapped into their VFPY equivalents. This includes handling constructs such as exceptions, Python classes, and method contexts. Throughout, we highlight key challenges and design decisions, supported by examples that illustrate how the translation system operates in practice.

5.1 Facts and Expressions

One can distinguish two specific types of components in Nagini and Verifast annotations.

- We define *facts* as *logical assertions that are assumed, derived, or checked to hold at a given point of the program*. These include pure facts (like mathematical truths, made of comparisons) and predicate facts (about memory or permissions).

  Multiple facts can be combined using the separating conjunction (`&*&` in VeriFast, and `and` in Nagini[1]), which asserts that they hold over disjoint parts of the memory state. However, facts cannot be used as conditions in ternary expressions or in disjunctions (neither in VeriFast

[1]Note that while this symbol is syntactically equivalent to the boolean conjunction, the user must keep in mind that they are semantically different.

nor in Nagini). Facts can also be combined in a conditional manner to form a fact again:

- Both VeriFast and Nagini support ternary facts

```
( cond1 ? fact1 : fact2)
```

```
fact1 if cond1 else fact2
```

- In addition, Nagini supports implications

```
Implies(cond1, fact1)
```

- *Expressions*, are used as building bricks in facts i.e. as operands in comparison or as argument in predicate instances or in functions calls (`fixpoint` functions in VeriFast, `@Pure` functions in Nagini). They can be of various types (integer, floating-point, boolean, etc...).

  Whereas boolean expressions are not directly considered as facts (and can thus be compared to boolean values or used as a conditions in ternary expressions), they can be raised to the level of facts, simply when used in a context in which a fact is expected[2].

Facts describe properties to verify; expressions are their building blocks. Understanding both is essential for correct translation. In the following subsections, we describe how these are modeled and handled by the automated translation process.

5.1.1 AST Translation

In order to distinguish both categories of nodes and properly translate them, our system relies on a recursive method (`is_predless`) traversing any Python Abstract Syntax Tree (AST) to determine by which process (fact or expression translation) it should be translated. Both processes translate a Python AST into a VeriFast/VFPY AST designed to be easily exported into VeriFast annotations.

The fact translation process delegates the translation of comparison operands, conditions of ternary expressions, and arguments of predicate instances to the expression translation process. Concretely, whenever translating a Python node, it goes through the following steps:

1. With the help of `is_predless`, it first detects whether the translation of the node should result in a *fact* or in an *expression*. It then delegates the translation of the node accordingly to either of the following methods:

[2]An exception in VeriFast is the case of a fixpoint function call such as $f(x_1, \ldots, x_n)$. In order to be used as a fact, such an occurrence must be compared with another boolean expression in order to be accepted as a fact.

- `translate_generic_fact`
- `translate_generic_expr`

2. Both methods mentioned in the previous point proceed to a case distinction on the type of the node. For each possible type, they will delegate the call to the corresponding method:
   - `translate_generic_fact`
     - for `Call`, call `translate_Call_fact`
     - for `IfExp`, call `translate_IfExp_fact`
     - for `BoolOp`, call `translate_BoolOp_fact`
   - `translate_generic_expr`
     - for `IfExp`, call `self.translate_IfExp_expr`
     - for `BoolOp`, call `self.translate_BoolOp_expr`
     - for `BinOp`, call `self.translate_BinOp_expr`
     - for `Compare`, call `self.translate_Compare_expr`.
       This method may rephrase some comparisons to bring them in a form that can be handled. For instance, a tuple comparison for equality `a==b` will be turned into a boolean conjunction `a[0] == b[0] and a[1] == b[1] and ...` while comparison for inequality will be turned in a disjunction of analogous inequalities.
     - for `Constant`, call `self.translate_Constant_expr`
     - for `Name`, call `self.translate_Name_expr`
     - for `Attribute`, call `self.translate_Attribute_expr`
     - for `Tuple`, call `self.translate_Tuple_expr`
     - for `Subscript`, call `self.translate_Subscript_expr`. This method will distinguish between Tuple indexing (only with constant indices)
     - for `Call`, call `self.translate_Call_expr`.
       In particular, this method will also handle expressions of the form `Implies(...,...)`, `Forall(...,...)`, `Old(...)`, `length(...)`, etc...
3. Each of the methods mentioned in the previous point will pass a context for VeriFast symbol translation to its callees. It will sometimes be the same as the context received as argument, but a context may be specifically created for the occasion in other cases.

5.1.2 VFPY Abstract Syntax Tree

The AST itself, now, is designed to represent *facts* and `expressions`. It therefore consists of multiple layers of concepts, of which a rough overview could be phrased as follows:

- Facts, as defined at the beginning of the chapter, can be of the following forms:
  - Boolean facts (also called "pure" facts) simply consist of a Boolean expression stated among the facts, thereby "raised" to the status of a fact.
  - Ternary facts consist of a boolean expression `c` and two other facts `f1` and `f2`. They are printed as `c ? f1 : f2`.
  - Predicate facts, actually denoting predicate instances, are represented as `p(e1, e2, ..., en)`, where `p` is the predicate name and `e1`, `e2`, ..., `en` are the expressions used as arguments for the predicate instance.
  - Fact conjunctions represent what is formally called the *separating conjunction*. When one or more *fact conjunctions* are conjoined, they are automatically merged into a single conjunction (their respective conjoined facts are extracted and conjoined together in a single fact conjunction).

    Example 5.1 *Facts conjunctions `(f1 f2 f3)` and `(f4 f5)` are conjoined together. Instead of forming the conjunction of conjunctions `((f1 f2 f3) (f4 f5))`, the AST node will detect this situation and automatically merge them into `(f1 f2 f3 f4 f5)`, which is semantically equivalent. This small optimization reduces the number of redundant brackets used in the final, which greatly improves readability.*

- Expressions can take the following specific forms:
  - An expression is said to be *name-defining* when it introduces a new symbol `x`, in which case the symbol is written `?x`. Such occurrences may only occur directly as arguments of a predicate instance, in an equality used as boolean fact, or in a constructor. *Name-defining* expressions are referred to as *patterns* in the VeriFast literature.

    Such expressions are represented by a class called `NameUseExpr`.
  - More commonly, an expression may also take one of the following forms:
    * name-using occurence: either part of the function arguments/return values, or introduced by pattern-matching in previous facts;
    * Literal: `5`, `5.5`, `'a'`, etc.;
    * Operation: these combine one or several sub-expression and thereby form a new expression. These may be of different types:
      · arithmetic

    - comparisons
    - boolean
    - fixpoint function calls
  * Constructor, corresponding to an immediate value (i.e. constructed in situ and not represented by a symbol) of a VeriFast inductive type. Some built-in examples are `some(...)`, `cons(...)`, `nil`, etc.

Each of the concepts enumerated here is represented as a class in our modeling of VFPY ASTs. Classes and helper methods have been defined in three modules:

- a module called `vf`, containing classes representing VeriFast native and built-in concepts (list, pairs, predicate, inductive types, operations, pattern-matching, etc...);
- a module called `pymodules` and often imported as `vfpy`, containing classes representing most the concepts discussed in chapter 4 (*hasval hasattr* and *hascontent* predicates, inductive types, etc...);
- a third module called `nag` [3], to separate Nagini-specific concepts from general Python concepts.

5.2 Symbol Translation

An essential difference between VFPY and Nagini derives from the duality of Python semantics (subsection 3.3.1).

Example 5.2 *Accessing a tuple item's value requires to go through one level of indirection and to introduce a new symbol. This is also the case for PyObject attributes or any other value wrapped in a Python Object.*

```
1 /*@ requires pyobj_hasval(args, PyTuple_v(
2         cons(pair(?arg1_ptr, PyLong_t)
3         cons(pair(?arg2_ptr, ***), nil))))
4     &*& pyobj_hasval(arg1_ptr, PyLong_v(arg1_val))
5     &*& pyobj_hasattr(arg2_ptr, "f", ?arg2_DOTf__ptr)
6     &*& pyobj_hasval(arg2_DOTf__ptr, PyFloat_v(?
    arg2_DOTf__val));
7 @*/
```

Having a structure supporting this translation therefore clearly appears as a necessity.

[3] The module `nag` is implicitly imported when importing `pymodules` and is never explicitly imported in client modules.

5.2.1 Context

A Python-to-VeriFast context (denoted `py2vf_context` in the code) is a mapping of Python expressions to VeriFast symbols. It is used to translate Nagini specifications into their VFPY equivalents.

Queries

In the following, we will denote a *context query*, or simply a *query*, as a lookup operation attempting to retrieve a specific symbol using the context object. Queries will be discussed more in detail later after some helper concepts. What matters in order to understand the next steps is that a query is made with:

- an Python AST node `n` (usually an instance of `ast.Name` or `ast.Call`);
- some additional values providing information about the *context* in which `n` is used;
- some additional parameters configuring the desired behavior of the lookup function.

The context's lookup function then returns either a name-defining expression, or a name-using expression.

Updates Throughout the Translation Process

A py2vf_context may be updated during the translation process. When a symbol is queried for the first time, the py2vf_context will return a name-defining expression (see name-defining expressions in 5.1.2) and cache the created symbol. The py2vf_context will return simple symbol-using expressions in the next queries.

Example 5.3 *The following specification setup describes a Python method taking one single integer argument* `i`*. Since the symbols denoting* `i`*'s pointer (*`i__ptr`*) and* `i`*'s wrapped integer value (*`i__val`*) are not natively defined in the specifications (as part of the C method arguments, for instance), the corresponding symbols must be introduced.*

```
1 PyObject* some_method(PyObject* self, PyObject* args)
    ;
2 /*@
3 requires pyobj_hasvalue(args, PyTuple_v(cons(pair(?
    i__ptr, PyLong_t), nil))) &*&
4 pyobj_hasvalue(i__ptr, PyLong_v(?i__val)) &*&//*** @
    */
5
```

Scope Differences and Parent Context

Not all VeriFast symbols introduced throughout the various assertions can be used in the same scope. Some may have been defined with a very wide scope, while others may have been defined with a very limited scope.

Example 5.4 *In the following specification,* `x_3` *and* `x_4` *are only defined in their respective branches. They are undefined outside those.*

```
1 int method(int x_1, int x_2);
2 /*@ requires bool_expr?
3         ( pred_instance(x_1, x_2, ?x_3) &*& x_3>=0 ):
4         ( pred_instance(x_2, x_1, ?x_4) &*& x_4<0 );
5 @*/ //@ ensures  true;
```

Consequently, it is of central importance not to let the translation process use symbols in a scope in which they are undefined. If such a case occurs, the translation process should raise an error and not write invalid specifications by returning a symbol undefined in that context.

In order to represent the hierarchy of scopes, we use several contexts (one for each scope) initialized in an analogous hierarchy: an instance `A` of py2vf_context can be initialized with a *parent context*: `A`'s parent context is another instance of *py2vf_context* representing (and containing the symbols defined in) the smallest scope that is strictly bigger than `A`'s scope.

Each context keeps track of the symbols defined exactly in its respective scope by maintaining its own dictionary. Upon a symbol query on `A`, `A`'s lookup function will turn the received query into a string value and try to retrieve the queried symbol from its own dictionary.

If nothing matches the query hash in `A`'s dictionary, the requested symbol may have been defined in a parent scope. Thus, `A` will delegate the query to its parent context and will recursively through the contexts representing the scope hierarchy until it reaches the context representing the top (widest) scope. If the queried symbol is defined in a scope wider than `A`'s scope (whose symbols can be queried using `A`'s parent), the delegated query will succeed and `A` will finally return the queried symbol.

Example 5.5 *In the specifications of a method, the top-most scope is the precondition's scope: indeed, symbols defined in the precondition can also be used in the postconditions.*

The postcondition's scope is already one step below (i.e. smaller): the symbol `result` *can be used in the postcondition, but not in the precondition.*

Old(...) Context

When translating `Old(...)` expressions, the translation process needs to use another context: it should stop translating mutable locations with symbols introduced in the *postcondition*'s scope, and should use the corresponding symbols introduced in the *precondition*'s scope instead.

In practice, this context switch is processed as follows:

1. the method in charge of translating the node extracts the content of the `Old(...)` expression.

$$\texttt{tr\_expr(n, ctx)} = \begin{cases} \texttt{tr\_expr(n.args[0], ctx.old)} & \text{if } \texttt{n} == \texttt{Old(...)} \\ \texttt{delegated\_tr\_expr(n, ctx)} & \text{otherwise} \end{cases}$$

Example 5.6 *Let us consider the following Python specifications.*

```
def method(x: classA) -> int:
    Requires(Acc(x.attr2))
    Ensures(Acc(x.attr2) and x.attr2 == Old(x.attr2))
```

They are automatically translated as follows

```
PyObject* method(PyObject* self, PyObject* args);
/*@
requires // *** SETUP defining x__ptr from args
pyobj_hasattr(x__ptr, "attr2", ?x_DOT_attr2__ptr) &*&
pyobj_hasvalue(x_DOT_attr2__ptr, PyLong_v(?
   x_DOT_attr2__val));
@*/

/*@
ensures // *** SETUP reusing x__ptr and args
pyobj_hasattr(x__ptr, "attr2", ?NEW_x_DOT_attr2__ptr)
    &*&
pyobj_hasvalue(NEW_x_DOT_attr2__ptr, PyLong_v(?
   NEW_x_DOT_attr2__val)) &*&
(NEW_x_DOT_attr2__val == x_DOT_attr2__val);
@*/
```

The last line shows the symbols respectively translated from values of `x.attr2` *and* `Old(x.attr2)`.

5.2.2 Delayed Access

In order to support symbol translation when using a context's lookup function, we defined a range of Python classes whose instances can be passed

as argument along with a the (Python Ast) node `n` to provide the lookup function with some details about the context in which `n` was used.

Example 5.7 *Upon translating an* `is` *expression, one does not know how high the AST of both operand expressions can be, i.e. how many steps one will have to go through before finally finding a variable name to translate.*

```
(a if bool1 else (b if bool2 else c)) is d
```

The above expression should translate into something close to the following:

```
//@ (bool1_val ? a_ptr : (bool2_val ? b_ptr : c_ptr))
    == d_ptr;
```

Thus, the expression `(a if bool1 else (b if bool2 else c))` *and then the expression* `(b if bool2 else c)` *must all be translated "keeping in mind" that the final result should be a reference, and not the value wrapped in these variables' PyObjects.*

The point of *delayed access* is to keep track of the context in which a node must be translated. We track this by passing an additional parameter of type `AccessType` in the expression translation process.

We defined `AccessType` is an abstract Python class that can be extended in different ways. With its subclasses, it aims to define an algebraic datatype (ADT) used to keep track of the way in which a value is accessed with respect to the corresponding symbol on which it is based. While `AccessType` and its subclasses are defined as Python classes, they can be conceptually understood as the following ADT:

```
    LeafAccess = PtrAccess | ValAccess
    AccessType = LeafAccess | AttrAccess(name: str,
  nested: AccessType) | Content(nested: AccessType)
```

Understanding in detail how they work is not central, to understand the functioning of symbol translation contexts. What matters is that

Example 5.8 *Consider the following specifications.*

```
#a: class A,
#a.f: in
#b: Tuple[int, float]
#x: object
Requires(Acc(a.t) and a.t==b[0] and a.t is not x)
```

For the above specification the following table illustrates the use of delayed access classes by describing the AccessType instance received as argument upon name translation (i.e. when translating the Nagini Name expression into a VFPY symbol).

Context	*Node*	*Name*	*Access Type Upon Name Translation*	*Resulting Symbol*
`a.t is x`	`x`	`x`	`PtrAccess()`	`x_ptr`
`a.t is x`	`a.t`	`a`	`AttrAccess("t",PtrAccess())`	`a_DOTt_ptr`
`a.t == b[0]`	`a.t`	`a`	`AttrAccess("t",ValAccess())`	`a_DOTt_val`
`a.t == b[0]`	`b[0]`	`b`	`TupleSbscAccess(0,ValAccess())`	`b_AT0_val`

The above symbol table allows the translation process to retrieve the correct symbols when translating the above Nagini specifications into VFPY:

```
/*@
requires //*** setup defining x_ptr and a_ptr ***
&*& pyobj_hasattr(a_ptr, "t", ?a_DOTt__ptr)
&*& pyobj_hasval(a_DOTt__ptr, PyLong_v(?a_DOTt__val))
&*& a_DOTt__val == 0
&*& a_DOTt__ptr == x_ptr;
@*/
```

5.2.3 Querying Contexts For Symbols

In order to query a context for a VeriFast symbol, the method `getExpr` must be called with two arguments:

- an expression `E` denoting either an occurence of an argument (i.e. an instance of `ast.Name`) or Result() (i.e. something of the form `ast.Call("Result", [])`.
- an instance of `AccessType` expressing what path must be applied on `E` to retrieve the wanted symbol.

Contexts support two kinds of queries that differ from each other in the way a query is handled if no corresponding symbol is found:

- read-only queries will raise an exception if the queried symbol cannot be found in the context itself or any of its ancestors;
- other queries will introduce a new symbol in the context and return a name-defining expression of the requested symbol (i.e. it will be printed with a leading question mark `?`).

Both semantics are distinguished by a keyword parameter `useonly` among parameters of `py2vf_context.getExpr`.

Example 5.9 *The signature of a symbol translation context's lookup function, taking arguments as described above.*

```
def getExpr(self, key: ast.Expr,
            AccessType: AccessType,
            useonly: bool = False)-> vf.NameOccurence
   :
```

5.2.4 Example: Context Initialization

This section illustrates how the context is initialized before translating symbols in VFPY.

Example 5.10 *The following example shows how the concepts introduced in this section can be used. This code illustrates the initialization of the context used for the postcondition translation.*

```
1  py2vf_ctx_postcond = py2vf_context(
2      parent=py2vf_ctx_setup,
3      prefix="NEW_",
4      old=py2vf_ctx_precond)
5  resultcall = ast.Call(
6      ast.Name("Result", ast.Load()),
7      [], [])
8  py2vf_ctx_postcond.setExpr(
9      resultcall,
10     PtrAccess(),
11     vf.NamedValue("result"))
12 py2vf_ctx_postcond.setprefix("NEW_")
13 #here translate postcondition
```

We emphasize following aspects:

- *The symbols defined in the setup assertions of the precondition (relating to the arguments) are still accessible in the postcondition and must not be redefined (this will cause a naming conflict otherwise). Thus, the context for postcondition translation takes the setup's context as parent context.*
- *Whenever an expression inside* `Old(...)`*-expression is translated, the translation process will query symbols from the context representing precondition's scope instead of the context representing the postcondition's scope.*
- *We then artificially create an AST node representing a call to* `Result()` *in a Nagini expression. This AST node is used to configure the postcondition translation context to translate* `Result()` *into VeriFast's built-in* `result` *when* `Result()` *is used as a reference.*
- *Finally, we set a prefix to be used for all symbols introduced in the postcondition translation. This prefix is used to avoid name conflicts with the symbols introduced when translating the precondition and to distinguish symbols when translating* `Old(...)` *expressions.*

5.3 Translating Specifications

This section details the process of translating Nagini specifications into VFPY-compatible VeriFast annotations. It covers how preconditions, postconditions,

and exception behaviors are expressed in the target format, ensuring that the semantics of the original Python specification are preserved for correct verification of the C implementation.

5.3.1 Setup

The *setup* of a specification denotes the part of specifications describing the method's arguments and introduces new symbols to refer to each argument's value. Even if the Nagini specifications are empty, the setup will be part of the VFPY translation.

In practice, the setup assertion's role is to include all facts introducing symbols that can be used to denote argument pointers and values when translating the specification itself. These symbols are added to a specific context during the construction of the setup assertion. By constructing, the setup also gives detailed information about the type of every argument.

The setup assertion of a method is always constructed as follows. For each parameter x of the Nagini method with name x_{name} and type x_{type}, following pair is added to the list of pairs carried by the constructor `PyTuple_v(...)`.

$$\texttt{pair}(x_{name}\texttt{\_\_ptr},\mathit{PyObj\_Type}(x_{type}))$$

In the above equation, the function *PyObj_Type* is defined as the translation function from Nagini types into VFPY PyObject types:

$$\mathit{PyObj\_Type}(x) = \begin{cases} \texttt{PyLong\_t} & \text{if } x = \texttt{int} \\ \texttt{PyFloat\_t} & \text{if } x = \texttt{float} \\ \texttt{PyUnicode\_t} & \text{if } x = \texttt{string} \\ \texttt{PyList\_t}(\mathit{PyObj\_Type}(y)) & \text{if } x = \texttt{List}(y) \\ \texttt{PyTuple\_t}(\mathit{PyObj\_Type}(y_1),\ldots) & \text{if } x = \texttt{Tuple}[y_1\ldots] \end{cases}$$

Example 5.11 *The following method with empty Nagini specifications still automatically gets a setup describing its arguments in its VFPY specifications (below):*

```
def test_setup1(i:int, f:float, b:bool) -> int:
    pass
```

```
PyObject* test_setup(self PyObject*, args PyObject*);
/*@
requires pyobj_hasvalue(args, PyTuple_v(cons(pair(?
   i__ptr, PyLong_t), cons(pair(?f__ptr, PyFloat_t),
   cons(pair(?b__ptr, PyBool_t), nil))))) &*&
pyobj_hasvalue(i__ptr, PyLong_v(?i__val)) &*&
pyobj_hasvalue(f__ptr, PyFloat_v(?f__val)) &*&
pyobj_hasvalue(b__ptr, PyBool_v(?b__val));@*/
```

In addition to the setup, the postcondition always contains additional assertions about the return value of the method.

5.3.2 PyExceptions

Our approach keeps things simple and requires the user to clear exceptions before leaving the function. This can be very easily expressed by adding a constant `PyExc(none, none)` to both the pre- and the postcondition.

5.3.3 Specification Body

After setting up the context and handling special cases such as exceptions, the main part of the specification must be translated. This subsection explains how the translation process converts the body of a Nagini specification into a VeriFast compatible format.

Expressions

Common pure expressions (e.g. arithmetic and boolean operations) from Nagini assertions are translated into VFPY, through a process turning the Python AST into a VFPY AST.

Let us recall here (from subsection 3.4.1) that we assume that specifications passed to our automated translation process must be well-defined in Nagini.

But some expressions that are well-defined in nagini will also be rejected and result in a translation failure[4]:

- calling a pure function in a context in which a reference is expected;
- any constant (e.g. `5`, `3.5`, `"My tailor is rich"` [2] or even `False`) used in a context in which a reference is expected;
- some operations existing in Python, but natively in C, like the exponent operator (e.g. `x**2` $\approx x^2$).

The next sections discuss other specific cases and limitations of the translation process.

Calls To Built-In Pure Functions

Calls to some native pure functions, like `len(l)` are automatically translated into the the VeriFast built-in fixpoint `length(l__content)`, where the symbol `l__content` denotes the list of pointer-values pairs of the list `l`.

Calls to other built-in Python pure functions(like `isinstance` and `issubclass`), however, have not been included in the automatic translation process, but

[4]The rationale for this choice is that these operations do not really have an immediate native equivalent in C or VeriFast. Some may however be easily translated.

one could very easily extend it and automatically translate them using the fixpoint functions `isinstance`, `issubclass` and `issubtype` defined in 4.5.3.

Attribute Access Permissions

Attribute Access Permissions (`Acc(x.f)`) are translated as as a separating conjunction of a `pyobj_hasval` instance and `pyobj_hasattr` instance, thereby allowing the user to refer to both the reference and the wrapped immutable value.

```
Requires(Acc(x.f))
```

```
/*@ requires
//setup defining x_ptr
    &*& pyobj_hasattr(x_ptr, ?x_DOTf__ptr)
    &*& pyobj_hasval(x_DOTf__ptr, ?x_DOTf__val);
@*/
```

When acquired fractionally only, the fraction must be propagated to the `pyobj_hasval` instance, as explained in subsection 4.4.2

```
Requires(Acc(x.f), 1/2)
```

```
/*@ requires
//setup defining x_ptr
    &*& [1/2]pyobj_hasattr(x_ptr, ?x_DOTf__ptr)
    &*& [1/2]pyobj_hasval(x_DOTf__ptr, ?x_DOTf__val);
@*/
```

Python List Access Permission

Python lists, like attributes, require explicit access permissions to be used safely in specifications or in the method's body. This subsection shows how list access is modeled in VFPY, including how both reference-level and value-level access to list items are expressed, and how the translation ensures proper handling of reference counting.

```
Requires(list_pred(l))
```

The above specification is then translated into the following VFPY specifications.

```
/*@ requires
    pyobj_hasvalue(args,
        Pytuple_v(cons(pair(?l__ptr, PyList_t(
   PyLong_t)), nil))
    &*& pyobj_hasvalue(l__ptr, PyList_v(PyLong_t))
```

```
5       &*& pyobj_hascontent(l__ptr, ?l__content__ptr)
6       &*& forall_predfact(?l__content,
7               pyobj_hasPyLongval, True, nil)
8       &*& (map(fst, l__content) == l__content__ptr)
9       &*& (some(map(snd, l__content)) ==
10              some(?l__content__val));
11  @*/
```

This allows the user to refer to the pointer of the `i`-th element with the expression `nth(i, l__content__ptr)`, and analogously for the value wrapped inside the `i`-th item using `nth(i, l__content__ptr)`, this equivalence is also embedded in the symbol translation system to translate `l[i]` in reference or value semantics.

However, the current state of the translation process for the `list_pred(l)` predicate is actually extremely limited.

- The automated translation process currently only supports one level of lists (our system does not handle variables of the type `List[List[int]]`, for instance);
- Constructing the iterated separated conjunction of `pyobj_hasval` instances on all list items is extremely complex to automate.

Ternary Facts

In the following, let `Z` be the context received as argument by the translation process in order to translate the whole ternary assertion.

In VeriFast, when a new symbol is introduced while translating a branch of some ternary fact, its scope is limited to that branch and it cannot be used outside. This must be taken into account when translating a ternary assertion `T` (phrased as `... if ... else ...`) from Nagini into VFPY.

In order to express the limited scope of symbols introduced during the translation of `T`'s branches (i.e., not pollute `Z`'s scope with new symbols introduced while translating assertions of `T`'s branches), a new context is created to translate each of them and takes `Z` as parent. This still allows the translation process to query previously introduced symbols, but will add nothing in `Z` that could interfere when translating assertions outside of `T`'s branches.

Each Nagini implication (written as `Implies(condition, assertion1)`) is translated analogously: as a VeriFast ternary assertion. The *if* branch of the ternary assertion contains the translation of the implicated assertion (`assertion1`) while the *else* branch of the is simply set as `true`.

Thus, not all Nagini implications or ternary facts can be translated into VFPY equivalents. Due to a bad phrasing, some specifications trigger translation problems, while they could have been correctly translated with a smarter form.

Example 5.12 *The following precondition is supported in Nagini, but the translation into VFPY fails:*

```
def fail_translation(x: A, y:A, z:bool)->int:
    Requires((Acc(x.f) if z else Acc(y.f)) and
            Implies(z, x.f==0.0))
    Ensures(...)
```

The first half of the assertion translates to the following assertion, but the introduced symbols cannot be used outside their respective branches. The translation does not succeed as the second occurrence of `x.f` *cannot be translated*

```
/*@ requires ... &*&
z__val ? (
pyobj_hasattr(x__ptr, "f", ?x_DOT_f__ptr) &*&
pyobj_hasvalue(x_DOT_f__ptr, PyFloat_v(?x_DOT_f__val)
   )) : (
pyobj_hasattr(y__ptr, "f", ?y_DOT_f__ptr) &*&
pyobj_hasvalue(y_DOT_f__ptr, PyFloat_v(?y_DOT_f__val)
   ))
```

A suitable form for this translation would rather be the following:

```
def fail_translation(x: A, y:A, z:bool)->int:
    Requires(((Acc(x.f) and x.f==0.0)if z else Acc(x.
   f)))
    Ensures(...)
```

5.4 Environment Translation

In addition to translating pre- and postconditions, verifying Python modules written in C requires translating the definitions of types, functions, and predicates. This section describes how these elements of the Python environment are represented in VFPY.

5.4.1 Class System

The Python class system actually does not need to be part of the VFPY global environment. However, for the sake of ease of use and readability, the translation process defines a fixpoint function for each user-defined class

that should make the hierarchy easier and less verbose to use, while preserving the construct allowing to use VFPY fixpoint functions like `isinstance`, `issubtype` and `issubclass`.

Finally, as modeled in section 4.5, an inductive `PyClass` value contains, beside its name, its superclass and a list of type arguments coming *in addition* to those of a generic superclass.

Example 5.13 *The following class hierarchy will be translated into the VFPY class hierarchy below.*

```
class Super(Generic[T, V]):
    pass
class Someclass(Generic[T, V, W], Super[T, V]):
    pass
```

```
/*@
fixpoint PyClass PyClass_ObjectType(){
    return ObjectType;
}
fixpoint PyClass PyClass_module_0Super(PyObj_Type T,
   PyObj_Type V){
    return PyClass("module_0Super",
   PyClass_ObjectType, nil);
}
fixpoint PyClass PyClass_module_0Someclass(PyObj_Type
    T, PyObj_Type V, PyObj_Type W){
    return PyClass("module_0Someclass",
   PyClass_module_0Someclass(T, V), cons(W, nil));
}
@*/
```

5.4.2 Pure Functions

This part describes how pure Python functions are translated into their VFPY equivalents.

Translation Process

The *Exprification* process denotes the transformation of a multiple-statement function body into a function with a single return statement, conceptually much closer to a VeriFast fixpoint function, defined as an expression in terms of the parameters, and therefore much easier to translate.

Exprification is greatly facilitated by the fact that pure functions supported by Nagini do not contain loops. Thanks to this, exprification mainly requires

compacting branches, variable assignments, and return statements into a single expression.

Example 5.14 *Let us consider the following Python/Nagini pure method.*

```
@Pure
def purefunction1(i: int) -> int:
    y = 18
    if (i > 0):
        y = y + 1
    elif (i < 0):
        return 0
    else:
        y = y * 2
        return y
    return y
```

The exprification process yields the following expression:

```
@Pure
def purefunction1(i: int) -> int:
    return 18 + 1 if i > 0 else 0 if i < 0 else 18 * 2
```

The exprified function can then be translated into a VeriFast fixpoint function:

```
/*@ fixpoint int PURE_purefunction1(PyObject * i__ptr
    , int i__val){
     return ((i__val > 0) ? (18 + 1) : ((i__val < 0) ?
     0 : (18 * 2)));
}@*/
```

For simplicity, we assumed as a baseline that the pure function's body could use each argument in reference or in value semantics. Each function parameter `x` in Nagini is therefore translated into two function parameters `x_ptr` and `x_val`, respectively denoting the PyObject reference and the immutable part of the value wrapped in the PyObject. A new context is initialized with all parameters before translating the exprified body of the pure function.

Example 5.15 *The major downside of the approach described above is that it strongly limits recursive numerical operations. Let us consider the following function:*

```
@Pure
def fibo(_n:int) -> int:
    return 1 if _n == 0 else\
           1 if _n == 1 else\
           1 if _n <0 else\
           fibo(_n-1) + fibo(_n-2)
```

The corresponding VFPY translation will have the following form.

```
/*@
fixpoint int PURE_fibo(PyObject* _n__ptr, int _n__val){
        return ((_n__val == 0) ? 1 :
               ((_n__val == 1) ? 1 :
               ((_n__val < 0) ? 1 :
               (PURE_fibo( ????, (_n__val - 1)) + PURE_fibo( ????, (_n__val - 2))))));
}
@*/
```

The question raised immediately is therefore: under the policy suggested above, what would one pass as a value to the first argument (expecting a PyObject pointer) (represented as `????` *above). Indeed, the result of a sum may still be used in reference semantics in Python, but things are obviously more complex in C.*

Consequently, we finally decided to use the following trick: whenever the name of a pure function argument will be prefixed with the label `VALUEONLY_` the translated fixpoint function's signature will only consider the value, omit the reference of the argument. Calls to the pure function are therefore be translated correspondingly by not passing the PyObject pointer of involved values, but only their wrapped value.

Example 5.16 *Taking advantage of this small feature now allows to translate our method.*

```
/*@
fixpoint int PURE_fibo(int VALUEONLY_n__val){
    return  ((VALUEONLY_n__val == 0) ? 1 :
            ((VALUEONLY_n__val == 1) ? 1 :
            ((VALUEONLY_n__val < 0) ? 1 :
            (PURE_fibo((VALUEONLY_n__val - 1)) + PURE_fibo((VALUEONLY_n__val - 2))))));
}@*/

```

More elaborate examples will be reviewed in the next chapter.

Precondition

While Nagini requires some access permission (e.g. `Acc(x.f)`) in a pure function's precondition in order for the function to be *allowed* to refer to a

mutable location `x.f`, VeriFast requires such access permission in order to be *able* to refer to the mutable location's content[5].

Since VeriFast fixpoint functions do not support any kind of precondition, it is even impossible to denote (because no symbol is introduced for it in the context) the value held by mutable locations. For this reason, pure functions with an impure precondition cannot be translated and are rejected by the translation process.

Return Types

In the current state of the work, we assume that pure functions are only used in value semantics and do not return references of any kind. Therefore, they can only return values of C native types or types built-in in VeriFast. Whenever a pure function is called in the specifications, the arguments of the corresponding fixpoint function are translated in a similar manner.

The main difficulty in pure function translation supporting more elaborate Python types lies in the fact that values returned by a VeriFast fixpoint function cannot be allocated in memory, like PyObjects normally are. This prevents us from reusing the PyObject modeling provided to specify API methods and would require a new such modeling to abstractly represent values wrapped inside PyObjects without relying on predicates or other permissions. This option will be discussed at the end of the report.

5.4.3 Predicates

Once assertions can be translated (section 5.1), translating a Nagini predicate P_N into a VFPY predicate P_{VFPY} is straightforward.

- The signature of P_{VFPY} contains two arguments for each argument of P_N': one denotes a pointer to the PyObject passed as argument, the other the immutable part of the value wrapped in that same PyObject.
- P_N's body can be translated into P_{VFPY} using the fact translation process, fed with a `py2vf_context` containing the parameters.

[5]Let us recall that if no access permission is acquired, no corresponding symbol is introduced, and it thereby becomes impossible to even refer to the value of the mutable location.

Chapter 6

Evaluation

In this chapter, we report on the various approaches chosen to put the developed translation process to the test. While the first section will focus on automated testing, the following sections will cover several case studies showcasing different features on which this work has been focusing: access to list items, access to attributes, and a wrapper function processing a numerical computation by delegating the work to a C library. The first two case studies are deliberately designed to remain simple and yield readable VFPY specifications while illustrating their respective topics; they do not necessarily have practical interest in real life. However, the last one is derived from a real-life module and will therefore require some adaptations to fit our needs and possibilities.

6.1 Unit Tests

Once a functionality has been properly developed and tested, it is central to ensure that introducing additional features and fixing pre-existing bugs will not break working functionalities. To this end, we wrote a complete battery of unit tests, allowing us to regularly re-check the consistency of the translation process' results.

Let us first define the most central terms used in this section. A *unit test*, in the most general sense, is a pair of a deliberately simple input and an *expected output* designed to verify a specific functionality of a program. Rather than simulations of real-world scenarios, these tests isolate and evaluate individual components to ensure that they behave as expected. The *result*, obtained by running the program on the *input*, is then compared to the expected output, to allow identification of nonconforming behaviors of the program.

In the next paragraphs, we discuss our approach to unit-test the translation process.

6.1.1 Oracle

First of all, one must keep in mind that the point of unit tests is not to demonstrate the code's correctness, but rather to ensure the code's stability throughout the updates: indeed, the expected result of all our unit tests has been produced by the translation process itself. Of course, each of our tests has been carefully reviewed several times[1], but expected results of our test do not come from an especially reliable oracle and should therefore be used *only* for consistency checks.

6.1.2 Practical aspects

Each input consists of a Python module (one file) whose name is prefixed with `test_`. When handed over to the translation algorithm described in chapter 5, the produced result is divided into several parts:

- a global VFPY environment representing the translated predicates and pure functions, as well as the class system;
- a pair of VFPY pre- and postconditions for every method of the module.

Each test input is designed to contain its expected output as a docstring[2]. Every method and every module has one, thereby allowing us to store the translated VFPY environment as the module's docstring, and the translated specifications of each method in the method's docstring.

Example 6.1 *The following code section displays an example of a unit test.*

```
"""
fixpoint PyClass PyClass_ObjectType(){
                return ObjectType;
}
"""
from nagini_contracts.contracts import *

def return_float() -> float:
        """
        requires PyExc(none, none) &*&
        pyobj_hasval(args, PyTuple_v(nil)) &*&
        true;
        ...
```

A specific module is in charge of running all the tests and interpreting the results. Upon running a test, the output is retrieved, whitespace-normalized[3]

[1] Some updates in the code are intended to impact the result, and therefore require an update of the affected unit tests.

[2] Documentation String. The Python `ast` library allows to easily retrieve it.

[3] Whitespace-normalization consists in removing tabulations, white spaces and line breaks. Remove inopportune indentations that spontaneously appear when formatting a test input in most common code editors.

and compared to the expected result. If the expected output and the result do not match, a list of differences between both (in their non-normalized form) is printed out.

Example 6.2 *The following terminal output illustrates how the dashboard displays the result and the output differences in case of failure.*

```
...
  ENV                   PASSED
  compare3              PASSED
Running test for file: test_arith.py
  ENV                   EMPTY
Running test for file: test_multimodules2.py
  ENV                   FAILED
        - Translation failed
        - Type error: Cannot find implementation or lib***
        - nagini_translation/native/tests/test_multimod***
        + fixpoint PyClass PyClass_ObjectType(){
        +           return ObjectType;
        + }
        + fixpoint PyClass PyClass_module_1mytupledclas***
        +           return PyClass("module_1mytupledcla***
        + }

Running test for file: test_pred.py
  ENV                   PASSED
  test1                 PASSED
  test2                 PASSED
...
```

6.2 Case Study: List Item Extraction

This case study demonstrates how VFPY can be used to verify a C implementation of a Python method that extracts an item from a list. We begin by specifying the method behavior in Python and then verify that the corresponding C code adheres to these specifications.

6.2.1 Python Specifications

We define the behavior of the list extraction method using Nagini specifications. The following extremelty simple method simpl retrieves the `i`-th item of the list `l` of length smaller than 100. The list is left unchanged.

```
def first_swap(l:List[int], i:int) -> int:
    Requires(list_pred(l) and len(l) > 0)
    Requires(len(l) > i and i >= 0 and len(l) < 100)
    Ensures(list_pred(l) and len(l) == Old(len(l)))
    Ensures(Result() is Old(l[i]))
```

```
6     Ensures(Forall(int, lambda j: Implies(j >= 0 and
    j < len(l), l[j] is Old(l[j]))))
```

The automatically translated VFPY specifications of the above can be found in section A.1

6.2.2 Verifying the Method

With the Python specifications translated into VFPY specifications, we now verify that the C implementation satisfies the latter one using VeriFast annotations.

```
1  List_extraction(PyObject *self, PyObject *args)
2  {
3      PyObject * l = PyTuple_GetItem(args, 0);
4      PyObject * i = PyTuple_GetItem(args, 1);
5      int i_val = PyLong_AsInt(i);
6      PyObject * l_i = PyList_GetItem(l, i_val);
7      //@ list_forallpred_extract(l__content,
     pyobj_hasPyLongval, i_val);
8      //@ open pyobj_hasPyLongval(l_i, ?l_i_val);
9      Py_INCREF(l_i);
10     //@ close pyobj_hasPyLongval(l_i, l_i_val);
11     //@ list_forallpred_insert(l__content,
     pyobj_hasPyLongval, i_val);
12
13     return l_i;
14 }
```

A short explaination of each of the code annotation may help the reader:

- Calling `list_forallpred_extract` on line 7 allows to extract the instance of `pyobj_hasPyLongval` on the `i`-th item of the list;
- `@open`-ing `pyobj_hasPyLongval` on line 8 yields an instance of `pyobj_hasval(l_i, PyLong` thereby satisfying the precondition rquired to call `Py_INCREF` on `l_i`;
- line 10 simply undoes `@open`-ing made on line 8;
- analogously, line 11 undoes the extraction processed on line 7, thereby already satisfy the postcondition.

These very simple annotations already suffice for VeriFast to successfully verify this method.

6.3 Case Study: Attribute access

This case study demonstrates how VFPY verifies C code that performs attribute access, starting from Python-level specifications and translating them into Verifiable C annotations.

Even though they remain extremely simple, the specifications in the following are already a bit more elaborate: the method takes a Python integer `i` and an instance `n` of the Python class A as arguments. Then replaces `n.y` with `i` and return the inital value of `n.y`.

6.3.1 Nagini Specifications

We define Nagini specifications for attribute access, outlining the required preconditions and postconditions that guide the verification of the corresponding C implementation.

```
class A:
    def __init__(self, y: int):
        self.y = y

@ContractOnly
@Native
def replace_and_get(n: A, i:int) -> int:
    Requires(Acc(n.y))
    Ensures(Acc(n.y) and  n.y is i)
    Ensures(Result() is Old(n.y))
```

The automatically translated VFPY specifications of the above can be found in section B.1

6.3.2 Verifying the Method

We specify the list extraction method in Nagini, describing its expected input and output behavior as a contract for the corresponding C implementation to fulfill.

Implementation

The following code shows the C implementation of the attribute access method, annotated with VFPY specifications to ensure correctness and proper reference handling.

```
Attr_extraction(PyObject *self, PyObject *args){
   PyObject * n = PyTuple_GetItem(args, 0);
   PyObject * i = PyTuple_GetItem(args, 1);
   const char * attr_name = "y";
```

```
5     PyObject * n_DOT_y;
6     n_DOT_y = PyObject_GetAttrString(n, attr_name);
7     while(n_DOT_y == NULL)
8     /*@invariant ... @*/{
9        n_DOT_y = PyObject_GetAttrString(n, attr_name);
10    }
11    PyErr_Clear();
12    int r = PyObject_SetAttrString(n, attr_name, i);
13    while(r == -1)
14    /*@invariant ... @*/{
15       PyErr_Clear();
16    }
17    return n_DOT_y;
18 }
```

The both while loops are necessary to ensure the following two elements.

- The attribute has been correctly retrieved. If an exception occurred, it is reflected by the NULL return value and the extraction is tried again until it succeeds
- Analogously, that the attribute has been correctly set. Retries until success in case of failure.

The two truncated loop invariants were already sufficient to prove the method's correctness. The complete loop invariants can be found in section B.2

6.4 Case Study: GMPY2's Binomial Coefficient Function

GMPY2 is a Python library designed for high-performance multiple-precision arithmetic[1]. It acts as a wrapper for the GMP (GNU Multiple Precision), MPFR (floating-point), and MPC (complex) libraries, offering fast and reliable computations with integers, rationals, floats, and complex numbers at arbitrary precision. It is particularly useful in applications requiring exact arithmetic or very large numbers (cryptography, numerical analysis, computational number theory).

In this section, we will consider how to verify *GMPY2*'s function for computing the binomial coefficient, `GMPY_MPZ_Function_Bincoef`.

6.4.1 Python Specifications

We begin the GMPY2 case study by specifying the expected behavior of the binomial coefficient function in Nagini. These specifications define the input

constraints and the guarantees the C implementation must fulfill.

Avoiding Overflow

In order to avoid handling overflow problems[4], we will assume (as part of the preconditions) that the handled numbers are small enough to keep the computed binomial coefficient in the range of values represented by native C `unsigned long long` type. We will use the trivial following inequality (directly derived from the binomial theorem) to bound the value of n:

$$\binom{n}{k} \leq 2^n \tag{6.1}$$

We will therefore assume $n \leq 63 \leq$ `sizeof(unsigned long long)` in the precondition, which therefore implies that the computation result will not suffer from an overflow:

$$\binom{n}{k} \leq 2^n \leq 2^{63} \leq \sum_{i=0}^{63} 2^i = 2^{64} - 1 = \texttt{ULLONG\_MAX}$$

Input of Specification Translation

We describe the structure and assumptions of the Nagini specifications that serve as input for the translation into VFPY-compatible VeriFast specifications.

```
@ContractOnly
@Native
def compute_bincoeff(n: int, k: int) -> int:
    Requires(n >= 0 and k >= 0 and k <= n and n <= 63)
    Ensures(Result() == bincoeff(n, k))
```

The uncommon phrasing of the body of`bincoeff` below is related to VeriFast's constraints on the definition of fixpoint functions[5].

```
@Pure
def bincoeff(VALUEONLY_n: int, VALUEONLY_k: int) -> int:
    Requires(VALUEONLY_n >= 0 and VALUEONLY_k >= 0 and
   VALUEONLY_n >= VALUEONLY_k)
    if VALUEONLY_k == 0:
        return 1
    elif(VALUEONLY_n == VALUEONLY_k):
        return 1
    else:
```

[4]Verifying a program handling overflow problems is a very interesting problem, and actually possible with the tools developed during this work. However, it makes the example considerably harder to follow and is therefore ill-suited for a case study.

[5]VeriFast typically requires the top-most expression of a fixpoint function to be either a switch or a ternary expression whose condition is an equality.

```
9           return bincoeff(VALUEONLY_n - 1, VALUEONLY_k - 1)
    + bincoeff(VALUEONLY_n - 1, VALUEONLY_k)
```

6.4.2 Translating Specifications into VFPY

Unsurprisingly, the VFPY translation of the specifications turns out to be extremely verbose. We therefore simplify them here by not showing the setup, since it does not show anything of interest in this case and would be the same as for any other function with the same signature.

```
static PyObject *
GMPY_MPZ_Function_Bincoef(PyObject *self, PyObject *args)
/*@
requires PyExc(none, none) &*&
//*** SETUP defining n__val and k__val ***

    (n__val >= 0) && (k__val >= 0) && (k__val <= n__val)
   && (n__val <= 63);

@*/
/*@
ensures PyExc(none, none) &*&
//*** SETUP reusing n__val and k__val ***

    pyobj_hasval(result, PyLong_v(?result__val)) &*&
    result__val == PURE_bincoeff(n__val, k__val);

@*/
{
...
```

The postcondition relies on the fixpoint function `PURE_bincoeff` declared in the file's global VeriFast environment, as a result of translating the corresponding pure function `bincoeff`.

Environment

The following code section sets up the environment required for verification, including class and function definitions used by the translated VFPY specifications for the GMPY2 function.

```
/*@
fixpoint PyClass PyClass_ObjectType(){
    return ObjectType;
}
fixpoint int PURE_bincoeff(int VALUEONLY_n__val, int
   VALUEONLY_k__val){
```

```
6       return ((VALUEONLY_k__val == 0) ? 1 : ((
      VALUEONLY_n__val == VALUEONLY_k__val) ? 1 : (
      PURE_bincoeff((VALUEONLY_n__val - 1), (
      VALUEONLY_k__val - 1)) + PURE_bincoeff((
      VALUEONLY_n__val - 1), VALUEONLY_k__val))));
7 }
8 @*/
```

6.4.3 Method Implementation

This section focuses on the practical adaptations made to the original implementation of the GMPY2 to verify it using VFPY. These changes include simplifying C constructs not supported by VeriFast, aligning the function interface with our modeling assumptions.

Simplifications

To make verification possible, several parts of the original code were simplified. These changes remove unsupported features and reduce complexity without affecting the core behavior of the function.

- GMPY defines its own type of PyObject and directly manipulates fields of such PyObjects. It thereby assumes that such PyObjects are only to be used by GMPY methods and that no other method should interact with those.

  Example 6.3 *The following example showing the last lines of the wrapper function* `GMPY_MPZ_Function_Bincoef` *of* `GMPY2` *emphasizes that the memory fragment of the returned PyObject(*`result`*) can be directly accessed.*

```
1 static PyObject *
2 GMPY_MPZ_Function_Bincoef(PyObject *self,
     PyObject * const *args, Py_ssize_t nargs)
3 {
4     ...
5     mpz_bin_ui(result->z, tempx->z, k);
6     Py_DECREF((PyObject*)tempx);
7     return (PyObject*)result;
8 }
```

  Additionally, GMPY uses a specific API to interact with its particular PyObjects. Some of the methods of that API delegate part of their work to methods of the Python API.

  Example 6.4 *The following code section was found in the implementation of* `GMPY_MPZ_Function_Bincoef`.

```
...
if (!(result = GMPY_MPZ_New(NULL))) {
    return NULL;
}
k = GMPY_Integer_AsUnsignedLong(args[1]);
...
```

We will want to verify a module working exclusively with conventional PyObjects and C native values, that directly relies on the API method. The initial implementation of `GMPY_MPZ_Function_Bincoef` has therefore been rewritten to a less efficient but simpler one.

- The syntax of C programs supported by VeriFast is limited and does not support all instructions and type declarations used in the original implementation of the method. We had to rephrase them to a form that is accepted by VeriFast.

  Example 6.5 *The type* `mpz_t` *of mpz values is declared as follows.*

```
typedef struct { ... } __mpz_struct;
typedef __mpz_struct mpz_t[1];
```

  VeriFast does not support array type definitions, which forced us to replace the definition as follows:

```
struct __mpz_struct { ... };
typedef struct __mpz_struct *mpz_t;
```

- The way in which methods of this module can be accessed from Python is configured in a specific manner, which induces a different signature than the methods supported by our specification translation process. The method's actual signature looks as follows:

```
static PyObject * GMPY_MPZ_Function_Bincoef(
    PyObject *self, PyObject * const *args, Py_ssize_t
    nargs)
```

- Finally, some checks performed by this function are obsolete when the function is considered with formal VFPY specifications. These checks have been removed.

  Example 6.6 *The following check becomes obsolete; not only because the method's signature does not include the number of arguments anymore, but because the number of arguments in our case*[6] *is fixed by the precondition.*

[6] Let us recall that we chose a configuration in which all arguments are wrapped in a single PyTuple object.

```
...
if (nargs != 2) {
    TYPE_ERROR("bincoef() requires two integer
   arguments");
    return NULL;
}...
```

Example 6.7 *The original implementation includes a check to detect overflows when converting an argument to an unsigned long. Since the translated version no longer uses GMPY-specific objects, this check must be adapted to rely on the standard Python API.*

```
...
k = GMPY_Integer_AsUnsignedLong(args[1]);
if (k == (unsigned long)(-1) && PyErr_Occurred())
{
    Py_DECREF((PyObject*)result);
    return NULL;
}
...
```

```
...
k = PyLong_AsUnsignedLong(args[1]);
if (k == (unsigned long)(-1) && PyErr_Occurred())
...
```

This check can be removed since we consider that the only exceptions that could be raised are `TypeError` *and* `OverflowError`*. The precondition of the* `PyLong_AsUnsignedLong` *requires conditions in which such exceptions cannot be raised: they clearly express the type of PyObjects wrapped in the PyTuple object received as an argument and the upper bound on the value it may contain.*

Despite all these changes, still verify wrapper code that performs a delegated call and uses the C `gmp` library to compute the binomial coefficient.

Final Implementation

This final version of the method includes all the changes discussed previously and has been fully adapted for verification in the VFPY system while preserving the original behavior.

```
static PyObject *
GMPY_MPZ_Function_Bincoef(PyObject *self, PyObject *
   args)
{
```

```
    struct __mpz_struct x_s = {0, 0, NULL};

    unsigned long n, k;
    n = PyLong_AsUnsignedLong(PyTuple_GetItem(args, 0));
    k = PyLong_AsUnsignedLong(PyTuple_GetItem(args, 1));

    mpz_t x = &x_s;
    mpz_init(x);
    mpz_bin_uiui(x, n, k);
    unsigned long res = mpz_get_ui(x);
    mpz_clear(x);

    PyObject *r = PyLong_FromUnsignedLong(res);

    while (r == NULL)
    {
        r = PyLong_FromUnsignedLong(res);
    }
    PyErr_Clear();
    return r;
}
```

6.4.4 MPZ Specifications

The goal of this case study is to verify a *wrapper* C extension. We will therefore consider the `mpz` library modularly as well as the Python API, and focus on verifying the code connecting both together.

We represent the access permission to a mpz value with the following predicate.

```
//@predicate is_mpz(mpz_t x; int val);
```

The core assumption here is again that only initialized mpz values should be passed to mpz methods for multiple-precision integer operations.

```
void mpz_init(mpz_t x);
/*@requires true;@*/
/*@ensures is_mpz(x,_);@*/
void mpz_clear(mpz_t x);
/*@requires is_mpz(x, _);@*/
/*@ensures true;@*/
```

The central `mpz` method used to compute the binomial coefficient is specified with a helper function. Note that the helper function has been deliberately designed with the exact same semantics as the Python pure `bincoeff` function.

```
/*@
fixpoint int bin_mpz(unsigned int n, unsigned int k){
  return ((k == 0) ? 1 :
          ((n == k) ? 1 :
            (bin_mpz((n - 1), (k - 1)) +
                bin_mpz((n - 1), k))
          )
         );
}
@*/
void mpz_bin_uiui(mpz_t res, unsigned long n,
   unsigned long k);
/*@requires is_mpz(res,_) &*& n >= 0 &*& k >= 0 &*& k
   <=n;@*/
/*@ensures is_mpz(res, ?res_val) &*& res_val ==
   bin_mpz(n, k);@*/
```

Finally, the function that allows one to extract the multiple-precision number represented in an mpz value is specified as follows. If the extracted number's representation is too big to fit in an `unsigned long long`, it will be truncated, and only the least significant bits will be returned. This can be expressed as `val & ULONG_MAX` since binary representation of `ULONG_MAX` is simply full of 1s.

```
unsigned long mpz_get_ui(mpz_t x);
/*@requires is_mpz(x, ?val);@*/
/*@ensures is_mpz(x, val) &*&
    result == ((val <= ULONG_MAX)?val:(val &
   ULONG_MAX));@*/
```

6.4.5 Verification

This section presents the final steps needed to complete verification: proving equivalence of numerical results, bounding results, and stating a loop invariant. It also discusses the limitations of the current system regarding termination proofs.

Equality of Fixpoint Functions

As can easily be checked, fixpoint functions `PURE_bincoeff` and `mpz_bin` have semantically identical definitions. However, VeriFast cannot derive

automatically that their results will consequently always be identical.

An easy way for the user to solve this is to help VeriFast by explicitly declaring such equivalences using an autolemma:

```
/*@
lemma_auto void
    bin_mpz_PURE_bincoeff(unsigned int n, unsigned int k);
  requires true;
  ensures PURE_bincoeff(n, k) == bin_mpz(n, k);
@*/
```

Another option would have been to specify `mpz_bin_ui_ui` using `PURE_bincoeff`, but this would have violated the modularity that we wish to preserve here: Indeed, the fixpoint function `mpz_bin` is defined to support the specifications of `mpz_bin_ui_ui` while `PURE_bincoeff` is automatically generated by our translation process to support the specifications of `compute_bincoeff`.

Binomial Coefficient And Overflow

Similarily, it is a trivial fact (derived from Equation 6.1) that for $n \leq 63$, the binomial coefficient will cause no overflow when stored in a variable of type `unsigned long long`. Again, VeriFast is unlikely to detect this fact, but an autolemma can again be written to help verify the program again.

```
/*@
lemma_auto void
    bin_mpz_bod(unsigned int n, unsigned int k);
  requires n >= 0 && k >= 0 && k <= n && n <= 63;
  ensures bin_mpz(n, k) >= 0 && bin_mpz(n, k) <=
   ULONG_MAX;
@*/
```

Loop Invariant

The last missing element before VeriFast successfully verifies the method's implementation is an invariant that ensures that the while loop will succeed. The loop invariant remains extremely simple, two important parts must be distinguished:

- The whole setup must be preserved as it is stated again in the postcondition;
- In addition, the permission representing the reference count increment of a successfully created object is only owned if the reference `r` is not null, which matches the specifications of `PyLong_FromUnsignedLong`. If `r` is null, an exception has been correspondingly raised, which is denoted by the equalities `e_ptr==some(_) &*& t_ptr==some(_)`.

```
1
2       while (r == NULL)
3   /*@
4       invariant PyExc(?e_ptr, ?t_ptr) &*&
5       pyobj_hasval(args, PyTuple_v(cons(pair(n__ptr,
        PyLong_t), cons(pair(k__ptr, PyLong_t), nil))))
        &*&
6       pyobj_hasval(n__ptr, PyLong_v(n__val)) &*&
7       pyobj_hasval(k__ptr, PyLong_v(k__val)) &*&
8       (r == NULL)?
9           e_ptr==some(_) &*& t_ptr==some(_):
10          pyobj_hasval(r, PyLong_v(res));
11  @*/
12      {
13          r = PyLong_FromUnsignedLong(res);
14      }
```

Termination

The acute reader will notice that this only proves partial correctness and neglects proving termination. Indeed, in the current setup, one cannot always guarantee termination and correctness against postconditions simultaneously. This dilemma is caused by the potential for methods from the Python API to fail and throw exceptions. In reaction to a failed method call, the client has several options.

- the client can ensure that the program does not fail by calling the method again after every failure, until the method finally succeeds;
- or the client can accept the failure and further propagate the exception back to Python by returning NULL.

However, the latter option is not supported by the current translation process, as the translation process never allows a method to return `NULL`, nor to terminate with a raised exception. This forces the client to call a method an unbounded number of times to ensure that it succeeds when the expected return value needs to be allocated[7].

[7] Immortal Objects like `Py_True`, `Py_False` or `Py_None` can do not need to be allocated, calling `Py_INCREF` on these will return an new owned permission without any risk of allocation failure.

Chapter 7

Future Works

This chapter presents possible directions for extending and improving the translation and verification framework.

- A first major improvement would be to redesign the symbol translation system to favor a more modular approach.

  One of the main difficulties during this work came from the attempt to refer to each value explicitly with one symbol name. This policy was manageable as long as the involved operands remained simple, but it quickly became more intricate when considering operands that vary with the method's arguments.

  For example, the Python expression `x[3].t[0]` would have been translated into a single name of the form `x_AT3_DOTt_AT0__ptr`, which is ill-suited for more complex cases that cannot be easily represented as a static name. Consider, for instance, how one would translate `x[i+1][2*j]` where `i` and `j` are function arguments.

  Using more advanced VeriFast constructs, one could attempt to introduce symbols and types to reach a form closer to `ptr(subsc(0, attr("t", subsc(3, x))))`, which is both easier to translate and more flexible for specifying complex expressions.

- A possible extension would be to support Nagini's abstract data structures, such as sets, maps, multisets, and sequences. Modeling these in VeriFast would allow the translation system to verify a broader range of Python specifications, particularly those involving high-level container logic or symbolic reasoning over collections.

- Another improvement would be to allow users to state assertions about C memory directly within the translated specifications. This would make it possible to preserve knowledge about initialized memory loca-

tions across function boundaries. Currently, VeriFast would treat such initialized memory as leaked if no assertion is provided.

- Finally, extending the current state of VFPY to support additional Python containers, such as dictionaries or sets, would open up many new opportunities.

Chapter 8

Conclusion

This thesis presented the first way to verify Python modules that are written in C. These modules are frequently used in real-world software because they offer improved performance or special features. But making sure they work correctly is difficult, especially since existing verification tools only focus on one language at a time. Our system, called **VFPY**, helps fill this gap. It connects the Nagini verifier for Python and the VeriFast verifier for C, making it possible to verify code that mixes both languages.

The core idea of this work is to take Python-style specifications (using Nagini) and translate them into VeriFast-style specifications for the C implementation. We also modeled relevant internal concepts of the Python runtime in VeriFast. This includes support for reference counting, exception handling, and dynamic typing, which are all important parts of Python's behavior.

We tested our system on several examples, including functions from real Python modules. These case studies show that our system can be used to verify that a C function behaves correctly when called from Python. The VFPY framework can be used in a modular way, meaning users can check one part at a time.

There are still limits to what our system can do. For example, it does not support all parts of the Python C API yet, and does not fully model all its features. In the future, it could be improved to support more built-in types, handle more complex specifications, or automate more parts of the translation. Even with these limits, this thesis lays a strong foundation for better tools that can help developers write safe and correct Python modules in C.

Appendix A

Annotations for Case Study "List Extraction"

A.1 Automatically Translated VFPY Specifications

```
PyObject *s List_extraction(PyObject *self, PyObject *args);
/*@
requires PyExc(none, none) &*&
pyobj_hasval(args, PyTuple_v(cons(pair(?l__ptr, PyList_t(PyLong_t)), cons(pair(?i__ptr, PyLong_t), nil)))) &*&
pyobj_hasval(l__ptr, PyList_v(PyLong_t)) &*&
pyobj_hasval(i__ptr, PyLong_v(?i__val)) &*&
pyobj_hascontent(l__ptr, List(?l__content__ptr)) &*&
list_forallpred(?l__content, pyobj_hasPyLongval, True, nil) &*&
(map(fst, l__content) == l__content__ptr) &*&
(some(map(snd, l__content)) == some(?l__content__val)) &*&
(length(l__content__val) > i__val) &*&
(i__val >= 0) &*&
(length(l__content__val) <= 100);
@*/
/*@
ensures PyExc(none, none) &*&
pyobj_hasval(args, PyTuple_v(cons(pair(l__ptr, PyList_t(PyLong_t)), cons(pair(i__ptr, PyLong_t), nil)))) &*&
pyobj_hasval(l__ptr, PyList_v(PyLong_t)) &*&
pyobj_hasval(i__ptr, PyLong_v(i__val)) &*&
pyobj_hasval(result, PyLong_v(?result__val)) &*&
pyobj_hascontent(l__ptr, List(?NEW_l__content__ptr)) &*&
list_forallpred(?NEW_l__content, pyobj_hasPyLongval, True, nil) &*&
(map(fst, NEW_l__content) == NEW_l__content__ptr) &*&
(some(map(snd, NEW_l__content)) == some(?NEW_l__content__val)) &*&
```

```
25 (length(NEW_l__content__val) == length(l__content__val))
      &*&
26 forall_(int j__val; (((j__val >= 0) && (j__val < length(
      NEW_l__content__val))) ? (nth(j__val,
      NEW_l__content__ptr) == nth(j__val, l__content__ptr)) :
       true)) &*&
27 (result == nth(i__val, l__content__ptr));
28 @*/
```

Appendix B

Annotations - "Attribute Extraction" Case Study

B.1 Automatically Translated VFPY Specifications

```
1  static PyObject *
2  Attr_extraction(PyObject *self, PyObject *args);
3  /*@ requires PyExc(none, none) &*&
4      pyobj_hasval(args, PyTuple_v(cons(pair(?n__ptr,
       PyClass_t(PyClass_module_0A())), cons(pair(?i__ptr
       , PyLong_t), nil)))) &*&
5      pyobj_hasval(n__ptr, PyClassInstance_v(
       PyClass_module_0A())) &*&
6      pyobj_hasval(i__ptr, PyLong_v(?i__val)) &*&
7      pyobj_hasattr(n__ptr, "y", ?n_DOT_y__ptr) &*&
8      pyobj_hasval(n_DOT_y__ptr, PyLong_v(?n_DOT_y__val
       )); @*/
9  /*@ ensures PyExc(none, none) &*&
10     pyobj_hasval(args, PyTuple_v(cons(pair(n__ptr,
       PyClass_t(PyClass_module_0A())), cons(pair(i__ptr,
        PyLong_t), nil)))) &*&
11     pyobj_hasval(n__ptr, PyClassInstance_v(
       PyClass_module_0A())) &*&
12     pyobj_hasval(i__ptr, PyLong_v(i__val)) &*&
13     pyobj_hasval(result, PyLong_v(?result__val)) &*&
14     pyobj_hasattr(n__ptr, "y", ?NEW_n_DOT_y__ptr) &*&
15     pyobj_hasval(NEW_n_DOT_y__ptr, PyLong_v(?
       NEW_n_DOT_y__val)) &*&
16     (NEW_n_DOT_y__ptr == i__ptr) &*&
17     (result == n_DOT_y__ptr); @*/
```

B.2 Loop Invariants

```
while(n_DOT_y == NULL)
/*@invariant
    PyExc(?e_ptr, ?t_ptr) &*&
    pyobj_hasval(args, PyTuple_v(cons(pair(n__ptr,
   PyClass_t(PyClass_module_0A())), cons(pair(i__ptr,
    PyLong_t), nil)))) &*&
    pyobj_hasval(n__ptr, PyClassInstance_v(
   PyClass_module_0A())) &*&
    pyobj_hasval(i__ptr, PyLong_v(i__val)) &*&
    pyobj_hasattr(n__ptr, "y", n_DOT_y__ptr) &*&
    pyobj_hasval(n_DOT_y__ptr, PyLong_v(n_DOT_y__val)
   ) &*&
    [f]string(attr_name, "y") &*&
    (n_DOT_y == NULL)?
        (e_ptr==some(_) &*& t_ptr==some(_)):
        (pyobj_hasval(n_DOT_y__ptr, PyLong_v(
   n_DOT_y__val))&*& n_DOT_y__ptr == n_DOT_y);
@*/
    n_DOT_y = PyObject_GetAttrString(n, attr_name);
```

```
while(r == -1)
/*@invariant
    PyExc(?e_ptr_2, ?t_ptr_2) &*&
    pyobj_hasval(args, PyTuple_v(cons(pair(n__ptr,
   PyClass_t(PyClass_module_0A())), cons(pair(i__ptr,
    PyLong_t), nil)))) &*&
    pyobj_hasval(n__ptr, PyClassInstance_v(
   PyClass_module_0A())) &*&
    pyobj_hasval(i__ptr, PyLong_v(i__val)) &*&
    pyobj_hasattr(n__ptr, "y", ?new_n_DOT_y__ptr) &*&
    pyobj_hasval(new_n_DOT_y__ptr, PyLong_v(?
   new_n_DOT_y__val)) &*&
    (r == -1)?
        (new_n_DOT_y__ptr == n_DOT_y__ptr):
        (new_n_DOT_y__ptr == i &*& e_ptr_2 == none
   &*& t_ptr_2 == none);
@*/
        PyErr_Clear();
```

Appendix C

Code Repository

The source code developed for this thesis is available at: `https://github.com/marcoeilers/nagini/tree/extract_native_specs`

The version corresponding to this document is identified by commit hash: `fad7666`, committed on April 24, 2025.

Bibliography

[1] Case Van Horsen. GMPY2: Multiple-precision arithmetic for Python. `https://gmpy2.readthedocs.io/`. Accessed: 2025-04-24.

[2] Alphonse Chérel. *L'anglais sans peine*. Assimil, 1930. Famous for the phrase "My tailor is rich", used in early English lessons.

[3] Marco Eilers and Peter Müller. *Nagini: A Static Verifier for Python: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, pages 596–603. 07 2018.

[4] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 41–55, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[5] Python Software Foundation. Extending and embedding the python interpreter. `https://docs.python.org/3/c-api/`. Accessed: 2025-04-24.

[6] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002.

[7] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. *Formal Methods in System Design*, 43(3):180–212, 2013.

[8] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. Pep 484 – type hints. `https://peps.python.org/pep-0484/`, 2014. Accessed: 2025-04-24.

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):

CROSS-LANGUAGE VERIFICATION FOR PYTHON MODULES WRITTEN IN C

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
BIRLING	ETIENNE

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)**
CHESEAUX-NORÉAZ, THU 24.04.25	

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