ETH zürich

Proving Python Code Correct with Nagini

Marco Eilers

Swiss Python Summit 2023

Who Am I?

- PostDoc at ETH Zurich, PhD in 2022
- **Working on Automated Program Verification**
  - **Python**
  - Security Properties

- Before: Two years in industry
  - Eclipse development
  - Domain Specific Languages
- Before that: Dual Study Programme at IBM

Python is great! But...

```
def usuallyWorks(s):
    decision = random.randint(0, 10000) > 5
    if decision:
      return "5" + s
    else:
      return 5 / s
```

```
usuallyWorks("test")
```

```
5test
```

```
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```


YOU COME TO ME AT RUNTIME
TO TELL ME THE CODE YOU ARE EXECUTING DOES NOT COMPILE
memegenerator.net

There is a Solution!

```python
def usuallyWorks(s: str) -> str:
    decision = random.randint(0, 10000) > 5
    if decision:
      return "5" + s
    else:
      return 5 / s
```

```
usuallyWorks("test")
```

```
Mypy: Unsupported operand types for / ("int" and "str")
```

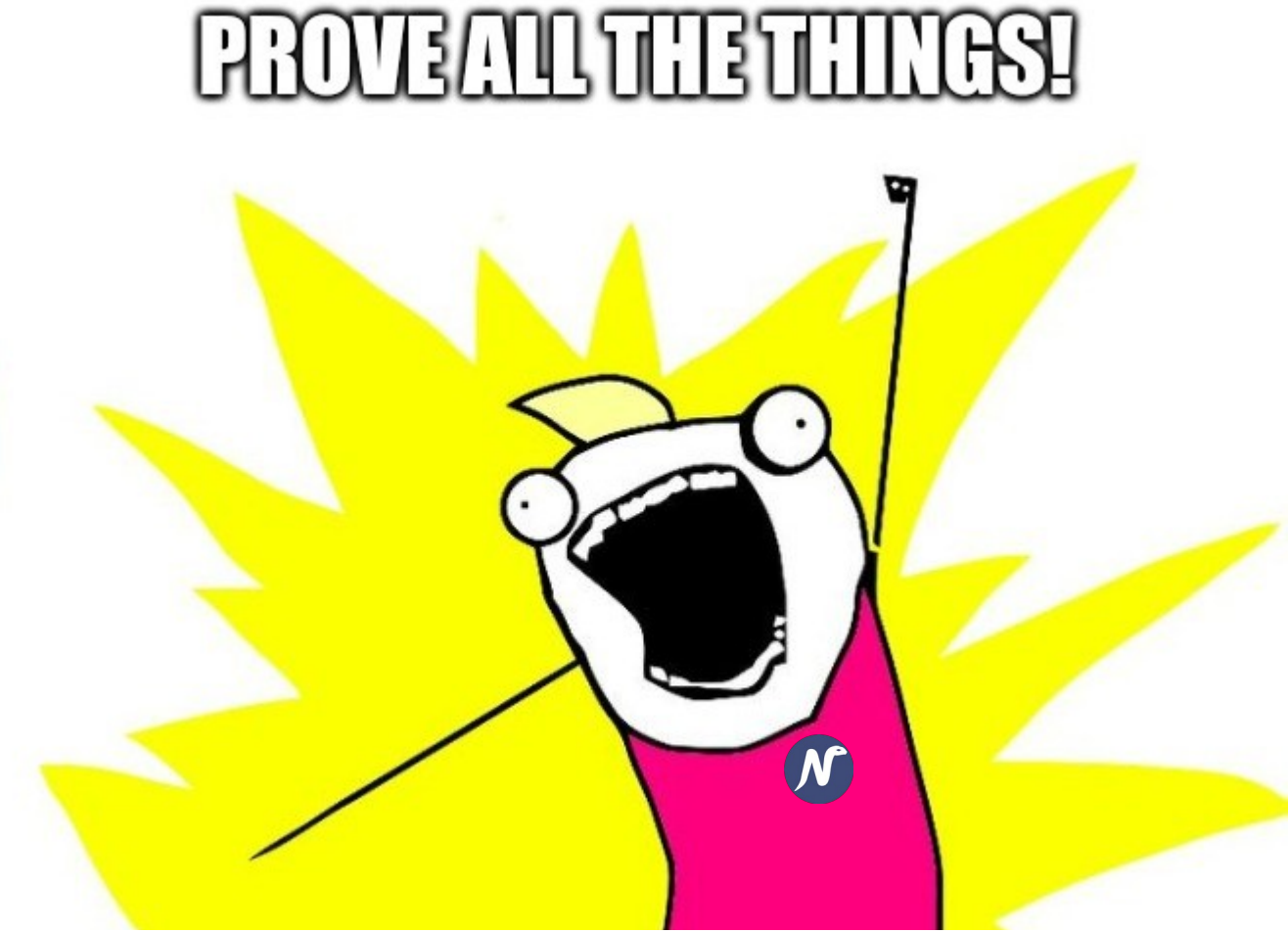
This talk: Why not go twelve steps further?

- Why stop at type errors?
  - Prevent runtime errors!
  - Ensure correct functional behavior!
  - Prevent unwanted information flow!

This talk: Why not go twelve steps further?

- Why stop at type errors?
  - Prevent runtime errors!
  - Ensure correct functional behavior!
  - Prevent unwanted information flow!

- Nagini: Mypy but way more general
  - Prevents all runtime errors by default
  - Prevents uncaught exceptions
  - Proves user assertions and specifications



Beyond Type Errors

```python
def usuallyWorks(s: int) -> int:
    decision = random.randint(0, 10000) > 5
    if decision:
      return 5 + s
    else:
      return 5 // s
```

```
usuallyWorks(0)
```

```
Mypy: Success: no issues found in 1 source file
```

```
ZeroDivisionError: division by zero
```

Demo

```
from nagini_contracts.contracts import *


def usuallyWorks(s: int) -> int:
  Requires(s != 0) # Add precondition using call to contract function
  decision = randint(0, 10000) > 5
  if decision:
    return 5 + s
  else:
    return 5 // s # Error before adding precondition (div by zero)


def client(i: int) -> None:
  usuallyWorks(i) # Error after adding precondition
```

Demo

```
from nagini_contracts.contracts import *
from nagini_contracts.obligations import MustTerminate
from typing import cast, List, Tuple, Union


class Cell:
  def __init__(self) -> None:
    pass

  def bar(self) -> None:
    pass


def test_cast(o: object, b: bool) -> int:
  Requires(Implies(b, isinstance(o, int))) # Flexible preconditions to ensure
  Requires(Implies(not b, isinstance(o, Cell))) # casts are safe
  if b:
    return cast(int, o) + 2  # Nagini proves casts succeed
  cast(Cell, o).bar()
  return 0
```

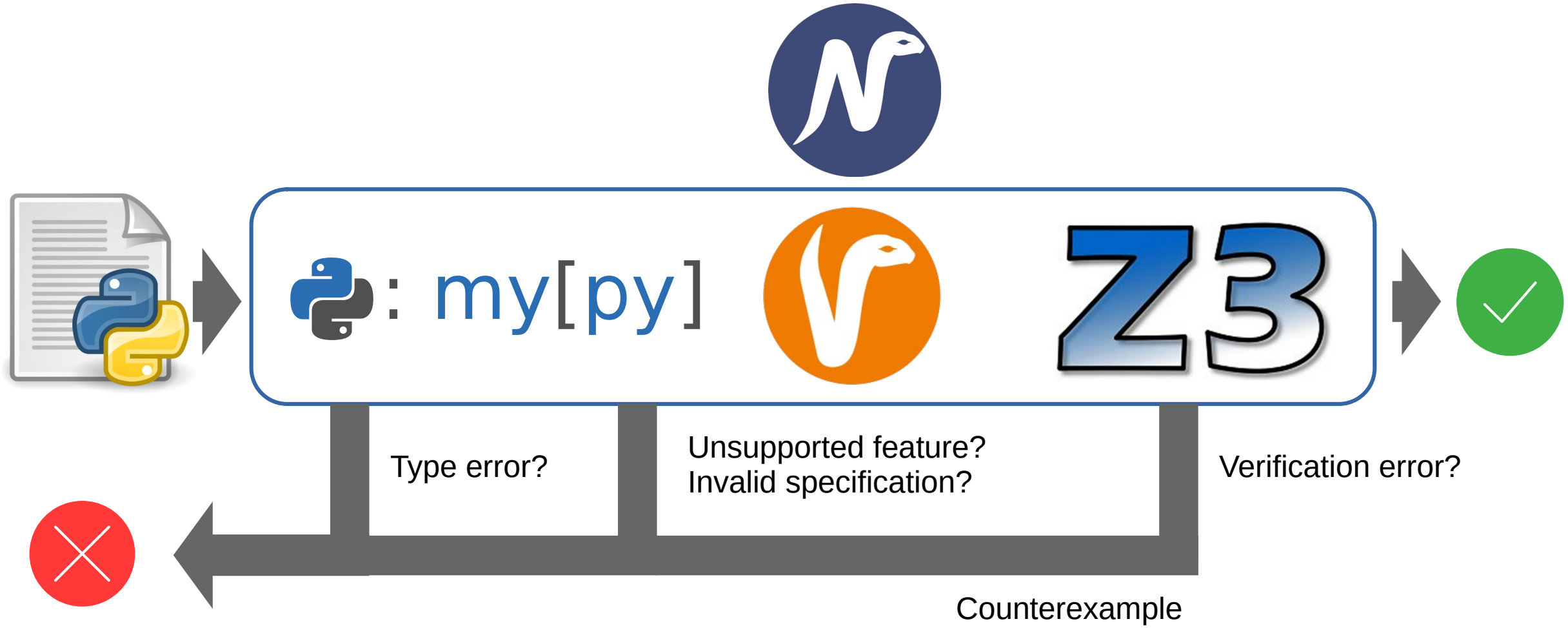
Demo

```
from nagini_contracts.contracts import *
from nagini_contracts.obligations import MustTerminate


def intsUntil(n: int) -> int:
  Requires(n > 0)
  Requires(MustTerminate(1)) # Prove termination
  Ensures(Result() == (n * (n - 1)) // 2) # Prove functional correctness

  sum = 0
  i  = 0
  while i < n:
    Invariant(i >= 0 and i <= n)
    Invariant(sum == (i * (i - 1)) // 2) # Invariants required
    Invariant(MustTerminate(n - i)) # to help the prover
    sum += i
    i += 1
  return sum
```

Demo

```
from nagini_contracts.contracts import *
from nagini_contracts.obligations import MustTerminate


def intsUntil2(n: int) -> int:
  Requires(n >= 0)
  Ensures(Result() == (n * (n - 1)) // 2)
  sum = 0
  for i in range(0, n):
    # Slightly different invariant with for loop
    Invariant(sum == (len(Previous(i)) * (len(Previous(i)) - 1)) // 2)
    sum += i
  return sum
```

Architecture



Demo: Counterexample

```
from nagini_contracts.contracts import *

def foo(i1: int, i2: int) -> None:
  if i1 == i2:
    Assert(i1 is i2)
```

```
Verification failed
Errors:
Assert might fail. Assertion (i1 is i2) might not hold. (ce.py@5.15--5.23).
Current store:
  i1 -> 0,
  i2 -> False
Current heap: Empty.

Verification took 5.84 seconds.
```

Demo: Quicksort

```
def quickSort(arr: List[int]) -> List[int]:
    Requires(Acc(list_pred(arr), 2/3))
    Requires(MustTerminate(2 + len(arr)))
    Ensures(Acc(list_pred(arr), 2/3))
    Ensures(Implies(len(arr) > 1, list_pred(Result())))
    Ensures(Implies(len(arr) <= 1, Result() is arr))
    Ensures(Old(ToMS(ToSeq(arr))) == ToMS(ToSeq(Result())))
    Ensures(Forall(int, lambda i: (Implies(i in Result(), Old(i in arr)), [[i in Result()]])))
    Ensures(Forall2(int, int, lambda i, j: (Implies(i >= 0 and i < j and j < len(Result()),
                                        Result()[i] <= Result()[j]), [[Result()[i], Result()[j]]])))
    less: List[int] = []
    pivotList: List[int] = []
    more: List[int] = []
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        j = 0
        while j < len(arr):
            Invariant(list_pred(less) and list_pred(pivotList) and list_pred(more))
            Invariant(Implies(j > 0, len(pivotList) > 0))
            Invariant(Acc(list_pred(arr), 1/2) and len(arr) > 0 and arr[0] == pivot)
            Invariant(j >= 0 and j <= len(arr))
            Invariant(ToMS(ToSeq(less)) + ToMS(ToSeq(more)) + ToMS(ToSeq(pivotList)) == ToMS(ToSeq(arr).take(j)))
            Invariant(Forall(int, lambda k: (Implies(k >= 0 and k < len(pivotList),
                                        pivotList[k] == pivot and pivot in arr), [[pivotList[k]]])))
            Invariant(Forall(int, lambda k: (Implies(k >= 0 and k < len(less), less[k] < pivot), [[less[k]]])))
            Invariant(Forall(int, lambda k: (Implies(k in less, k in arr and k < pivot), [[k in less]])))
            Invariant(Forall(int, lambda k: (Implies(k >= 0 and k < len(more), more[k] > pivot), [[more[k]]])))
            Invariant(Forall(int, lambda k: (Implies(k in more, k in arr and k > pivot), [[k in more]])))
            Invariant(Forall(int, lambda k: (Implies(k in pivotList, k in arr), [[k in pivotList]])))
            Invariant(MustTerminate(len(arr) - j))
            i = arr[j]
            if i < pivot:
                less.append(i)
            elif i > pivot:
                more.append(i)
            else:
                pivotList.append(i)
            tmp = ToSeq(arr).take(j) + PSeq(i)
            Assert(tmp == ToSeq(arr).take(j + 1))
            j += 1
        Assert(ToSeq(arr).take(j) == ToSeq(arr))
        less = quickSort(less)
        more = quickSort(more)
        Assert(Forall(int, lambda i: (Implies(i in less, Old(i in arr)), [[i in less]])))
        Assert(Forall(int, lambda i: (Implies(i in more, Old(i in arr)), [[i in more]])))
        Assert(Forall(int, lambda i: (Implies(i in pivotList, Old(i in arr)), [[i in pivotList]])))
        res = less + pivotList + more
        Assert(Forall2(int, int, lambda i, j: (Implies(0 <= i and 0 <= j and i < len(less) and j < len(pivotList),
                                        less[i] in less and less[i] < pivotList[j]), [[less[i], pivotList[j]]])))
        Assert(Forall2(int, int, lambda i, j: (Implies(0 <= i and 0 <= j and i < len(more) and j < len(pivotList),
                                        more[i] in more and more[i] > pivotList[j]), [[more[i], pivotList[j]]])))
        Assert(Forall2(int, int, lambda i, j: (Implies(0 <= i and 0 <= j and i < len(less) and j < len(more),
                                        less[i] in less and more[j] in more and less[i] < more[j]), [[less[i], more[j]]])))
        Assert(Forall(int, lambda i: (Implies(i >= 0 and i < len(res),
                                    res[i] is (less[i] if i < len(less) else (pivotList[i - len(less)] if i < len(less) + len(pivotList)
                                    else more[i - len(less) - len(pivotList)])), [[res[i]]])))
        return res
```

Caveats (1/2)

- Nagini is an academic prototype.
  - It will crash sometimes
  - A lot of features are supported:
    - Subclassing, overrides, exceptions, iterators, concurrency, properties and classmethods, generics, union types, ...
  - For some features, support is missing or incomplete
    - Lambdas, nested anything, match-case, many small things
  - Some dynamic features are intentionally not supported
    - Magic methods, metaclasses, eval, …
    - These are just difficult to reason about
  - More stubs for libraries required
  - Performance can be an issue
  - Documentation :(

Caveats (2/2)

- Specification is complex and requires a lot of work
  - You can verify that neat algorithm you wrote. You usually cannot verify your entire application.
  - Specifications are usually (a lot) longer than the code itself.
  - Verifying entire applications requires years of work by experts.
- Expert knowledge often required
  - Special reasoning constructs used for many complex language concepts
    - Framing, subtyping, concurrency, expressing complex properties
- Theoretical limits:
  - Automated reasoning:
    - Non-linear arithmetic is undecidable.
    - Floating point is incredibly slow.
    - Quantifiers work well until they don’t.

Future directions?

- More dynamic code support
  - ETH MSc student just started
- Less overhead to prove simple properties
  - Trying to achieve this on the Viper level
- Frequently-used libraries
  - Machine learning?
    - Not my area
    - Also: Non-linear

https://github.com/marcoeilers/nagini