Introduction to Horus — Part 1

Julian Sutherland

Follow

Nethermind.eth

--

Listen

Share

By

[Agustín Martinez Suñé

](https://www.linkedin.com/in/agusms/), thanks to

[František Silváši

](https://www.linkedin.com/in/frantisek-silvasi/),

[Greg Vardy

](https://twitter.com/0xgreg_?lang=en)and

[Julian Sutherland

](https://twitter.com/JulekSU)for reviewing and editing.

A few weeks ago, we announced the alpha release ofHorus. Horus is an open-source automated formal verification tool to help Starknet developers detect bugs in their smart contracts before they're even deployed. We'd like say a big thank you to StarkWare for supporting Horus during its initial development.

With Horus, you can annotate your code with formal specifications using a simple assertion language similar to Cairo. Horus will then try to verify that the contract's implementation meets these specifications. To understand how this works in practice, we're starting a blog series where we'll dive into Horus' key features, and learn how to verify an ERC20 smart contract and a toy AMM implementation.

Please note: Horus is currently in alpha; bug fixes and new features are being continuously merged. For the sake of stability, the following steps should be performed using

[this

](https://github.com/NethermindEth/horus-compile/tree/0484a10cf1e528737db24ebe4193c3dccd1b89a4)version of the

horus-compile

tool, and

[this

](https://github.com/NethermindEth/horus-checker/tree/8e3abaea6fada0037d55d20802c81896c9c9e2a1)version of the

horus-check

tool.

Program correctness 101

Let's consider the following simple Cairo program that increments its input value by 5:

We want to verify if it's correct or has any bugs. The standard way of gaining confidence in the correctness of your program is to come up with a multitude of test cases…

…and write them down using a testing framework. Then, the testing framework will execute your program for each of the inputs in your test suite and compare the actual output

with the expected output

.

So, what do you know after the execution of the tests?

1. If there is a mismatch between the actual output

and the expected output

, in which case you probably just caught a bug!

1. If there is no mismatch, you know that the program behaves correctly, at least for the set of inputs you provided in the test suite. However

, you still don't know how the program behaves for other inputs you haven't tested.

This is where formal verification comes into play; it allows you to mathematically prove

that a program is correct for any possible input

.

“

Program testing can be used very effectively to show the presence of bugs but never to show their absence

.” — Edsger Dijkstra

For a formal verification tool like Horus to analyze your program, it is your responsibility to state the program's intended behaviour

. You can think of it the following way: when you are writing a test case, you have a mental model of what the program is supposed to do;

that's how you can say the expected output is 7 if the input was 2.

Instead of writing specific examples, formal verification requires you to explicitly write down this mental model

— the intended behaviour of the program — in what is called a formal specification

.

Postconditions

The most basic part of a function specification is its postcondition, a logical formula stating what conditions should hold after the successful execution of the function

. We will use the following postcondition to specify the expected output of the function:

We used the keyword @post

, part of the Horus annotation language, to indicate we are about to write a postcondition. The keyword $Return

allows us to access the returned tuple and its field res

. As you can see in the annotations, we can also access the input values, such as x

.

Once again: we are not stating the expected output for a specific value of x

but for any possible value of x

. If you passed the annotated program above to Horus, you would get the following output:

This means our function is correct — it returns x + 5

for any possible x

passed as input! Hooray!

You might not be impressed by this tiny "of course, it adds five" example. But what if your function is a lot longer and more

complex? What if it has calls to other functions? Wouldn't it be great to simply write a declarative

postcondition that summarises the expected behaviour

and have an automated tool check the function complies with it

? That's exactly what Horus is about! What's more, usually, the description of the intended behaviour of a function can be put in much simpler terms than the instructions that compute such behaviour.

So what happens if the program is not correct?

Horus lets us know:

Currently, Horus doesn't give a counterexample in this situation, that is, concrete input values and an execution trace that shows the postcondition breaks, but that feature is in the roadmap, so keep your eyes open!

Branches

How does Horus deal with branches? Consider yet another function that just adds five to its input:

As expected, if we pass this annotated program to Horus, we'll get the following:

The difference with respect to previous examples is that now Horus has to check that @post

holds for any of the branches. It will also collect and make use of the information in the branch conditions in its attempts to prove the @post

. That's how it can conclude that the line return (res=7);

satisfies $Return.res == n + 5

, given that n == 2

.

But what if our program had a bug in one of its branches?

Horus will show a more verbose output with an explicit verdict for each branch:

Here, :::1

is Horus' way of saying "the True

branch", and :::2

"the False

branch". If there were nested if

s you'd see outputs like :::11

, :::12

, and so on.

As expected, the branch returning res=n+5

gets Verified

, and the branch returning res=8

gets False

.

Preconditions

Sometimes we write a function that assumes it will only be called for certain input values but not for others. For instance, in the context of Starknet's Automated Market Maker example, there are two possible token types: 0 or 1. The following helper function takes a token type and returns the opposite one.

In the @post

we are using boolean operators to express the possible scenarios and constrain the output value $Return.t

to the input value token

. But if we check this with Horus, we'll get the following output:

The else

branch fails to get verified, why is that? The problem is that both the code and the postcondition assume that the input value token

will be either 0 or 1. Here is where the precondition

enters the stage, a logical formula stating what conditions should hold right at the beginning of the execution of the function

.

We used the keyword @pre

to indicate we are about to write a precondition.

If we check this new version with Horus, we'll get the following:

What is this output telling us? That for any call to the function where token

is 0 or 1, the return value will surely be the opposite token.

More generally, a program, C, is said to be correct with respect to a precondition, P, and a postcondition, Q, if and only if, any final state resulting from a terminating execution of C starting in a state satisfying P must satisfy Q. This is usually denoted {P} S {Q}. If you didn't know about all this, congratulations — you just learned the very basics of Hoare Logic!

The power of formal verification in general, and of Horus in particular, is that it can allow us to mathematically verify that such specifications hold.

Function calls

Consider the following function that calls the get_opposite_token

function from the previous example:

The function foo

calls get_opposite_token

and then returns 42

, but when calling get_opposite_token

it passes 2

as the argument, which is not allowed. Since we provided a specification for get_opposite_token

(in the form of @pre

and @post

), Horus will use this information when verifying foo

's correctness. Horus will check if the precondition of get_opposite_token

is satisfied at the call site, and if it doesn't, foo

will fail to be verified:

The first two lines of output here resulting in False

is Horus checking if the precondition of get_opposite_token

holds before the call.

Instead, if the call to get_opposite_token

does satisfy its @pre

, Horus will assume its @post

holds right after the call, and continue reasoning about the verification of the program using this information, which is what the second result relating to foo

above indicates. In this case, since the function does return 42

as the specification requires, the verification goes through. The following example illustrates a more interesting situation where the verification depends on the postcondition of the function called:

The function bar

calls get_opposite_token

two times and returns the result. Note that get_opposite_token

's precondition holds both times because of bar

's own @pre

. Therefore, right after each call, Horus will know that the postcondition of get_opposite_token

holds.

Since we are calling get_opposite_token

once with token

and once with the output of the call, we'll end up getting the same initial value, which leads to $Return.res == token

.

Horus automatically does all this reasoning for us:

Recursion

We now have everything in place to discuss a more interesting example. The following function takes an arbitrary number m

and adds to it a non-negative number n

. Of course, we couldn't just do it with the +

operator for this blog post! We want to show you how the power of Horus can even be used to verify recursive functions:

Overall, the function works by taking m

and adding 1

per recursive call.

- The base case is add(m, 0)

, in which case, we just return m

.

- The recursive case is add(m,n)

with n > 0

. For that, we recursively call add(m, n - 1)

, which we know will be one step closer to the base case, and then add 1

to the recursive result.

If we pass this annotated program to Horus, we'll get the following:

Great! But how did Horus do it? The reasoning is actually similar to the way one thinks about recursive functions.

We already talked about how Horus deals with branches, so we can see that verifying the base case follows from assuming the branch condition (n == 0)

in order to prove that return (res=m);

satisfies $Return.res == m + n

.

The question is, how does it deal with the recursive call in the else

branch? Well, just like any other function call! Since we have provided a formal specification for the function in the form of a @pre

and a @post

we've immediately enabled Horus to reason about a recursive call:

- It will check if @pre

holds right before the function call. It does hold because, since we are in the else

branch, we know n > 0

, therefore n - 1 >= 0

.

- It will assume @post

holds right after the function call, which leads to the knowledge that res1 == m + n - 1

. This is exactly what's needed to conclude, after the instruction let res2 = res1 + 1

, that res2 == m + n

. QED

😄

Finally, this is the output we would get if one of the branches had a bug:

Horus' output:

Just like the branching example that we saw in the previous section!

That's it! You just learned about Horus' fundamentals and can start verifying your contracts! In Part 2, we'll introduce Horus' annotations for reasoning about storage variables and introduce the so called "logical variables" using the toy AMM from the "Hello, StarkNet" tutorial. Stay tuned!

The Horus alpha release supports Cairo 0.10.1, with a Cairo 1.0 release coming soon.

New to Starknet and looking for more developer tools? Check out Nethermind'sWarp — the Solidity to Cairo compiler,Juno — Starknet client implementation, and Voyager, a StarkNet block explorer!