Artifact for the OOPSLA 2024 paper "Hypra: A Deductive Program Verifier for Hyperproperties"

Introduction

This document describes the artifact for the OOPSLA 2024 paper “Hypra: A Deductive Program Verifier for Hyperproperties”, which consists of:

- Our tool Hypra.
- Our evaluation, with instructions to replicate it.
- An Isabelle/HOL proof of the soundness of the novel loop rule described in section 4.2 (Theorem 1), as well as Lemma 1.

Hardware Dependencies

The artifact is a VirtualBox VM image with Ubuntu 24.04 LTS that contains our tool Hypra, all benchmarks used in our evaluation, Isabelle 2024, and our Isabelle/HOL formalization. It uses 8GB of RAM and two cores by default; if these values are too high for your system, feel free to adjust them, but the VM may not work correctly.

Getting Started Guide

In the following, we explain:

1. How to open the VM
2. How to run Hypra on two examples
3. How to reproduce the evaluation, with 1 repetition only
4. How to check the Isabelle formalization

Opening the VM

To run the VM, simply import it into an up-to-date version of VirtualBox (we tested with version 7.0). The username is **test** and the password is **test**.

Running Hypra

To use Hypra to verify a program, type the command `hypra <path_to_program>` in the terminal. By default, Hypra generates both overapproximation and underapproximation encodings for the input program, emits overapproximation framing, verifies loop invariants modularly, and does not automatically select loop rules when verifying loops.

To test that Hypra can run correctly, type the following in a terminal to run Hypra on a program that should **not** verify:
> `hypra ~/hypra/src/test/evaluation/forall/descartes/time_false.hhl`

The expected output is as follows:

```
The input program is read from
/home/test/hypra/src/test/evaluation/forall/descartes/time_false.hhl
Parsing successful.
Type checking successful.
Translated program is being verified by Viper.
Carbon has been started to verify the program.
Silicon has been started to verify the program.
Carbon failed to verify the program:
Postcondition of compare1 might not hold. Assertion !(in_set_forall(_s1,
S): Bool) || !(in_set_forall(_s2, S): Bool) || (!((get(_s1, i): Int) ==
1) || !((get(_s2, i): Int) == 2) || (get(_s1, res): Int) == -1 *
(get(_s2, res): Int)) might not hold. (<no position>)
Silicon failed to verify the program:
Postcondition of compare1 might not hold. Assertion (forall _s1:
State[Int], _s2: State[Int] :: { (in_set_forall(_s1, S): Bool),
(in_set_forall(_s2, S): Bool) } { (in_set_forall(_s1, S): Bool),
(get(_s2, i): Int) } { (in_set_forall(_s1, S): Bool), (get(_s2, res):
Int) } { (in_set_forall(_s2, S): Bool), (get(_s1, i): Int) } {
(in_set_forall(_s2, S): Bool), (get(_s1, res): Int) } { (get(_s1, i):
Int), (get(_s2, i): Int) } { (get(_s1, i): Int), (get(_s2, res): Int) }
{ (get(_s2, i): Int), (get(_s1, res): Int) } { (get(_s1, res): Int),
(get(_s2, res): Int) } !(in_set_forall(_s1, S): Bool) ||
!(in_set_forall(_s2, S): Bool) || (!((get(_s1, i): Int) == 1) ||
!((get(_s2, i): Int) == 2) || (get(_s1, res): Int) == -1 * (get(_s2,
res): Int))) might not hold. (<no position>)

Verification failed
Postcondition of compare1 might not hold. Assertion !(in_set_forall(_s1,
S): Bool) || !(in_set_forall(_s2, S): Bool) || (!((get(_s1, i): Int) ==
1) || !((get(_s2, i): Int) == 2) || (get(_s1, res): Int) == -1 *
(get(_s2, res): Int)) might not hold. (<no position>)
Runtime: 8.708574176
```

Next, type the following command to run Hypra on a program that should verify:
> `hypra ~/hypra/src/test/evaluation/forall-exists/orhle/gni/denning1.hhl --auto`

The option `--auto` in this command tells the verifier to automatically select a loop rule when verifying the loop in the program.

The expected output is as follows:

```
The input program is read from
/home/test/hypra/src/test/evaluation/forall-exists/orhle/gni/denning1.hh
l
Parsing successful.
Type checking successful.
Carbon has been started to check a side condition.
Silicon has been started to check a side condition.
Carbon succeeded in verifying the side condition
Can use sync rule? true
Applying syncRule
Translated program is being verified by Viper.
Carbon has been started to verify the program.
Silicon has been started to verify the program.
Carbon succeeded in verifying the program
Verification succeeded
Runtime: 8.29586005
```

Running the Evaluation with One Repetition

To run the evaluation with 1 repetition, type the following command in the terminal (which should take a few minutes):
> `hypra_test 1`

The parameter “1” in the command above can be replaced with an arbitrary positive integer n to run the evaluation with n repetitions.

The following is the expected output of the command `hypra_test 1`:

```
Number of repetitions: 1
Evaluation No. 0 starts
src/test/evaluation/forall/descartes/defaultcookie.hhl OK
src/test/evaluation/forall/descartes/fileitem_false.hhl OK
src/test/evaluation/forall/descartes/namecomparator_false.hhl OK
...
---------------------
Total: 84
Failed: 0
Runtime: 257.95726495799994 s
Test data is saved to: src/test/evaluation/output0.csv
```

If Hypra produces the expected verification result for a benchmark, “OK” is printed next to the name of the benchmark; otherwise, “Failed” and the reason for failures are printed instead. At the end of the n-th repetition, a summary of the evaluation result is printed on the terminal, and a .csv file, named *output0.csv*, which contains all information of the evaluation, is created in the directory ~/hypra/src/test/evaluation.

To automatically obtain the median and average verification time of the entire benchmark suite, type the following command in the terminal after running the evaluation with at least 1 repetition:

> `cd ~/hypra; python3 process_data.py src/test/evaluation`

This produces a *result.csv* file located in the directory *~/hypra/result*. The expected output of the commands above can be found in *~/hypra/result/expected_result.csv*.

Checking the Isabelle formalization

Our Isabelle formalization builds on the one for Hyper Hoare Logic. Our contribution can be found in the file *AutomatingHHL.thy*.

To get started, we recommend making sure that all the files are successfully verified by Isabelle.

Our mechanization is located in ~/artifact/mechanization, and it contains the following 11 Isabelle files (10 are from the Hyper Hoare Logic formalization):

- PaperResults.thy
- Language.thy
- Logic.thy
- ProgramHyperproperties.thy
- SyntacticAssertions.thy
- Loops.thy
- Expressivity.thy
- Compositionality.thy
- ExamplesCompositionality.thy
- TotalLogic.thy
- **AutomatingHHL.thy (our contribution)**

a. Using Isabelle's CLI

One can check that Isabelle successfully verifies all 11 files using the Isabelle command line interface (located at *~/Isabelle2024/bin/isabelle*, accessible in the terminal via the alias *isabelle*) with the command *"isabelle build -c -d. -l HyperHoareLogic"* (this command tells Isabelle to build the *HyperHoareLogic* session, which is defined in the *ROOT* file), run from the folder ~/artifact/mechanization.

This can be achieved with the following command:

> `cd ~/artifact/mechanization`
> `isabelle build -c -d. -l HyperHoareLogic`

Expected output:

The final lines of the output should look like the following:

```
...
Session Unsorted/HyperHoareLogic
  /home/test/hypra/mechanization/AutomatingHHL.thy
  /home/test/hypra/mechanization/Compositionality.thy
  /home/test/hypra/mechanization/ExamplesCompositionality.thy
  /home/test/hypra/mechanization/Expressivity.thy
  /home/test/hypra/mechanization/Language.thy
  /home/test/hypra/mechanization/Logic.thy
  /home/test/hypra/mechanization/Loops.thy
  /home/test/hypra/mechanization/PaperResults.thy
  /home/test/hypra/mechanization/ProgramHyperproperties.thy
  /home/test/hypra/mechanization/SyntacticAssertions.thy
  /home/test/hypra/mechanization/TotalLogic.thy
Running HyperHoareLogic ...
Finished HyperHoareLogic (0:01:07 elapsed time, 0:01:47 cpu time, factor
1.61)
0:01:14 elapsed time, 0:01:47 cpu time, factor 1.45
```

This output indicates that Isabelle successfully verified the 11 files in 1 minute and 14 seconds (it might take a bit longer depending on your configuration). A different output might indicate a problem.

b. Using Isabelle's GUI

Isabelle's graphical user interface can also be used to ensure that Isabelle can verify all files. It is located at *~/Isabelle2024/Isabelle2024*, and can be opened by typing the command "isabelle_gui" in the terminal.

To check that our Isabelle formalization is successfully verified:

1. Open the file (File > Open…) *~/artifact/mechanization/AutomatingHHL.thy*, which contains the claims made in the paper as well as some explanations (at the top of the file).
2. Open the Theories panel (Plugins > Isabelle > Theories panel). It should be visible on the right of the window.
3. Activate "continuous checking" by ticking the box at the top of the Theories panel, if it is not already activated.
4. Put the cursor at the end of the file.

The verification status can be seen on the right of the editor, next to the scrollbar:

- Pink indicates a part that has not been verified yet.
- Purple indicates ongoing verification.
- Clear or orange indicates successful verification. Orange indicates a warning (warnings do not indicate invalid proofs, but correct proofs that can be optimized).
- Red indicates an error (this should *not* happen).

Moreover, the “Theories” panel shows the verification status of all files that are imported by the opened file, which need to be checked before the opened file can be checked.

To jump to the definition of a term, click on it while holding the Control key.

Step by Step Instructions

In this section, we first explain how our benchmarks are classified, and how we obtained them. We then describe minor changes that we made to the tool and evaluation since the submission. Finally, we explain how to reproduce the evaluation.

Benchmarks

The benchmarks used in our evaluation are taken from the benchmark suites of 4 state-of-the-art verifiers:

- Descartes (https://github.com/marcelosousa/descartes)
- ORHLE (https://github.com/rcdickerson/orhle)
- HyPa (https://github.com/hypa-tool/hypa/tree/main)
- PCSat (https://github.com/hiroshi-unno/coar)

The criteria of benchmark selection can be found in Section 5 of our paper.

All benchmarks used in our evaluation can be found in the directory *~/hypra/src/test/evaluation.* The path and the name of each benchmark express the type of hyperproperty that it reasons about, the source of the original benchmark, and whether it should verify or not:

- If a filename **ends** with "**_false**", then verification is expected to **fail**.
- If a filename **does not end** with "**_false**", then verification is expected to succeed.

Example 1: Benchmark *~/hypra/src/test/evaluation/forall-exists/hypa/counter_diff.hhl*
It requires verifying a $\forall^*\exists^*$-hyperproperty. It has been adapted from a benchmark of HyPa. It should verify since its name does not end with "false".

Example 2: Benchmark *~/hypra/src/test/evaluation/exists/orhle/backjack_once_false.hhl*
It requires verifying an $\exists^*$-hyperproperty. It has been adapted from a benchmark of ORHLE. It should not verify since its name ends with "false".

Translating the Benchmarks

At the beginning of each of our benchmarks, we have specified the location of the original benchmark. In most cases, the translation from the original benchmark to our Hypra benchmark is straightforward. The exceptions to this are listed below:

- For Descartes benchmarks, each of them is translated to a Hypra program with 3 methods that only differ in the specifications. Each of these methods reasons about a distinct hyperproperty, so the entire Hypra program can reason about all 3 hyperproperties that the original Descartes benchmark reasons about.
- Some HyPa benchmarks repeat a block of code **infinitely often**. However, Hyper Hoare Logic (on which Hypra is based) can only expresses hyperproperties over **terminating executions**. Thus, to faithfully model those examples, we modeled an **arbitrary** repetition of the code block (e.g., by forgetting the values of variables modified by this block of code).

- For the benchmark named half_square_ni from PCSat, by analyzing its semantics and specifications, we believe that the variable `y` should be incremented by `i` instead of by `y` in the loop body. This adjustment is reflected in our translation.
- For **invalid** benchmarks (i.e., where verification failure is expected), we only translated those without loops, because it is unclear whether examples with loops fail because the hyperproperty does not hold, or because the loop invariant is too weak.

Obtaining valid ∃* and ∃*∀ benchmarks from invalid ∀* and ∀*∃* benchmarks

We also obtained benchmarks for ∃* and ∃*∀*-hyperproperties by taking **invalid** Descartes and ORHLE benchmarks, i.e., benchmarks that do not satisfy some ∀* or ∀*∃*-hyperproperties (i.e., verification with Descartes or ORHLE fails). We changed their specification to **formally prove that they indeed violate the relevant hyperproperty**. To do so, we strengthened the preconditions and proved the negation of the original postconditions [Theorem 5, Dardinier and Müller 2023].

As a concrete example, the file *~/hypra/src/test/evaluation/forall/descartes/contact_false.hhl*, which is expected to **not verify**, is a translation of the file https://github.com/marcelosousa/descartes/blob/master/benchmarks/pldi-16/stackoverflow/Contact-false.java. This example **violates** (for instance) transitivity (P2), a **∀∀∀-hyperproperty**, and thus verification is expected to **fail**.
From this **invalid** benchmark, we additionally created the **valid** benchmark *~/hypra/src/test/evaluation/exists/descartes/contact_false_exists.hhl*, in which we formally prove that transivity is violated (by strenghtening the precondition and negating the postcindition), i.e., we prove an **∃∃∃-hyperproperty**.

Minor changes since the submission

We have made the following changes to the implementation and evaluation of Hypra after the initial paper submission:

- When Hypra is asked to automatically select loop rules to verify loops, it now generates a separate Viper program to evaluate the condition of applying syncRule/syncTotRule. Based on the result of this Viper program, Hypra decides whether to apply syncRule/syncTotRule or other loop rules. Previously, this condition was encoded as a Viper expression, and a Viper if-else statement was used to handle both true and false scenarios. As a result, the generated Viper program contained lots of nested code blocks, which made it extremely difficult to read and debug.
- We have removed 1 benchmark[1] from the benchmark suite. This benchmark fixes the value of a program variable. It is redundant because a more generalized version of this benchmark, where the program variable can take all possible values, already exists (*~/hypra/src/test/evaluation/forall/pcsat/double_square_ni.hhl*).

[1] Available at: https://github.com/hiroshi-unno/coar/blob/299e979bfce7d9b0532586bfc42b449fd0451531/benchmarks/pfwnCSP/cav2021rel/DoubleSquareNI_hFF.clp

Reproducing the Evaluation

To reproduce our evaluation, type in the following command to run the evaluation for 3 repetitions (as in the paper):

> `hypra_test 3`

The code performing this command can be found at *~/hypra/*src/main/scala/viper/HHLVerifier/test/Test.scala. It calls the `main` function in the file *~/hypra/src/main/scala/viper/HHLVerifier/Main.scala, where the runtime measurement starts immediately before the input program is being parsed (line 44) and ends immediately after the verification result becomes available (line 89).*

After this command (which lasts ~15 minutes in our experience), the results should be stored in the files ~/hypra/src/test/evaluation/output0.csv (for the first repetition), ~/hypra/src/test/evaluation/output1.csv (for the second repetition), etc.

To obtain the summary results, first copy the output files produced by each repetition (i.e., *output0.csv*, *output1.csv*, *output2.csv* located in *~/hypra/src/test/evaluation*) into a directory (e.g., ~/hypra/data) without other .csv files, for example by running the following command:

> `rm -rf ~/hypra/data; mkdir ~/hypra/data`
> `cp ~/hypra/src/test/evaluation/output*.csv ~/hypra/data`

To obtain the summary results, run (if you directory is ~/hypra/data):

> `cd ~/hypra; python3 process_data.py ~/hypra/data`

This should produce a *result.csv* file located in *~/hypra/result*. Open it and compare the mean and median runtime with the data in Table 1. Note that the *result.csv* file does not include the information about the number lines of program code or annotations, because this information was obtained manually.

Notice that most data of verification time in Table 1 are different from those in Figure 11 of our paper. This is caused by the changes stated above. The updated evaluation results will be included in the revised version of our paper. Note that the following data of verification time are collected on a MacBook Pro running macOS Ventura 13.3 with a 2.3 GHz 8-Core Intel Core i9 processor and 32 GB RAM.

Type of hyperproperties	Source	no.	Files Mean (LoC)	Verification time		Annotations Mean (LoC)
				Mean (s)	Median (s)	
∀*	Descartes	15	129	2.25	1.62	0
	PCSat	3	24	1.09	1.10	3
	Overall	18	112	2.06	1.54	1

∃*	Descartes	8	81	12.97	5.09	0
	ORHLE	6	29	2.89	2.54	8
	Overall	14	59	8.65	3.51	4
∀*∃*	ORHLE	28	21	2.79	2.02	2
	HyPa	8	14	1.18	1.11	3
	PCSat	1	22	2.14	2.14	2
	Overall	37	19	2.43	1.87	2
∃*∀*	ORHLE	15	25	2.66	1.99	2

Table 1. Updated results of our evaluation.

Reusability Guide: Using Hypra on New Examples

Writing a Hypra program

To create a new Hypra program, make a new file with **.hhl** as its extension. You may use any text editor to write the program, but **vim** is highly recommended since it provides syntax highlighting for Hypra programs.

In the following subsections, we show the syntax of Hypra programs. The formal big-step semantics of most commands can be found in Appendix A of Dardinier and Müller [2023].

Method declarations

Method declarations are the only top-level declarations allowed by Hypra. They:

- Are declared by the keyword `method`
- Have 0 or more input (e.g. `x`) and output (e.g. `z`) parameters of type `Int`
- Can have 0 or more hyper-assertions as preconditions
- Can have 0 or more hyper-assertions as postconditions
- Preconditions cannot talk about the set of erroneous states, as it is assumed to be empty at the beginning of each method, which allows modular reasoning of method calls

A method with precondition P, postcondition Q and body C verifies if and only if the hyper-triple {P} C {Q} is valid (see Section 2 of our paper).

```
method foo(x: Int, y: Int) // parameter(s)
returns (z: Int, w: Int) // return value(s)
requires … // precondition P
ensures …// postcondition Q
{
    // method body C
}
```

Hyper-assertions

A hyper-assertion is, informally, a quantified assertion over sets of program states (see Section 2 of our paper).

```
forall <_s> :: A
forall <<_s>> :: A
forall _i: Int :: A

exists <_s1>, <_s2> :: A
exists _i1: Int, _i2: Int:: A

// Examples:
```

```
var x: Int
forall <_s1> :: exists <_s2> :: _s1[x] >= _s2[x] // Valid hyper-assertion
forall _i: Int :: true // Not a hyper-assertion but a normal assertion,
                          because it does not quantify over states
```

- They should include one or more `forall` and/or `exists` quantifiers, with at least one of the quantifies over program states
- The variables that are being quantified over should have an identifier starting with **an underscore** and a letter, followed by zero or more letters and/or numbers
- Use the syntax `<_state_name>` to declare a normal program state that is being quantified over. For instance, `<_s>` declares a state `_s` that is in the current set of normal program states
- Use the syntax `<<_state_name>>` to declare an erroneous program state that is being quantified over. For instance, `<<_s>>` declares a state `_s` that is in the current set of erroneous program states
- Use the syntax `_var_name: Int` to declare an integer variable that is being quantified over
- The body of the hyper-assertion, `A`, is either a hyper-assertion or a boolean expression
- In a hyper-assertion, it is possible to get the value of a program variable by using the syntax `_s[v]`, where `_s` is a normal or erroneous program state being universally or existentially quantified over and `v` is a program variable

Boolean expressions

- Constants `true` and `false`
- Conjunction `e1 && e2`, disjunction `e1 || e2` and implication `e1 ==> e2` where `e1` and `e2` are both boolean expressions
- Equality `e1 == e2` and inequality `e1 != e2`, where `e1` and `e2` are both arithmetic expressions or both boolean expressions
- `e1 > e2`, `e1 >= e2`, `e1 <= e2` and `e1 < e2` where e1 and e2 are both arithmetic expressions
- Normal assertions with quantifiers `forall` and `exists` (They should only quantify over integers but not states, otherwise they are hyper-assertions)

Arithmetic expressions

- Constants of integer values
- Program variables of type `Int`
- Values of program variables in a state (e.g. `_s[v]`, see the part about hyper-assertions above)
- Variables that are quantified over by `forall` and `exists`
- Addition `e1 + e2`, subtraction `e1 - e2`, multiplication `e1 * e2`, division `e1 \ e2`, modulus `e1 % e2` where e1 and e2 are both arithmetic expressions

Variable declarations

- They are declared with keyword `var`
- The variable identifier must start with a letter, followed by 0 or more letters or numbers, e.g. v1
- Program variables can only have **integer** values at the moment

```
var v1: Int
```

`assume` and `assert` statements

- `assume` statements filter out program states where the boolean expression `e` does not hold.
- `assert` statements are similar to `assume` statements, except that program states where `e` does not hold will be collected as erroneous states.

```
assume e  // e is a boolean expression
assert e
```

`hyperAssume` and `hyperAssert` statements

Both `hyperAssume and hyperAssert` statements are useful for debugging programs

- `hyperAssume` statements add hyper-assertions to the program as additional assumptions
- `hyperAssert` statements add hyper-assertions to the program as additional assertions

```
hyperAssume A  // A is a hyper-assertion
hyperAssert A
```

Conditional statements

Conditional statements divide a set of states into two groups based on the truth value of the boolean expression `b` in the states, and then execute command `C1` in states where `b` holds and execute command `C2` in states where `b` does not hold. Note that the `else` branch is optional.

```
if (b)  // b is a boolean expression
{
   // command C1
} else { // The else branch is optional
   // command C2
}
```

Deterministic assignments

In each program state, a deterministic assignment `v := a` evaluates the arithmetic expression `a` in the state and assigns its value to the program variable `v`. Note that the variable `v` cannot be a method argument, since method arguments cannot be reassigned to.

```
v := a // v is a program variable, e is an arithmetic expression
```

Non-deterministic assignments

To assign a non-deterministic value to a program variable `v`, use the syntax shown below.

As explained in Section 2.1 of our paper, hints can be declared with the syntax `<hint_name>` as part of the non-deterministic assignments. They are useful for constructing witnesses, which can be achieved with `use` statements, during underapproximation reasoning. Note that the identifiers of hints must follow the same rule as the identifiers of program variables.

```
havoc v // v is a program variable
havoc v <h> // h is a hint
```

`use` statements

To construct witnesses during underapproximation reasoning, use the previously declared hints with `use` statements as shown below. The statement `use h(a)` tells Hypra to construct a witness state where the variable `v`, previously assigned non-deterministically with hint declaration `h`, is set to the value of the arithmetic expression `a`.

```
use h(a) // h is a hint, a is an arithmetic expression

// Example
var x: Int
havoc x {hint1}
use hint1(2) // This construct a witness state where x is 2
```

Loops

Besides a loop guard `b` and a loop body `C`, a loop should also have one or more loop invariants, at most one `decreases` clause and at most one rule annotation.

```
while rule (b) // b is a boolean expression
invariant A // A is a hyper-assertion
decreases a // a is an arithmetic expression
{
      C // loop body
}
```

Hypra handles a loop as follows:

- Before the loop, the loop invariants, which are hyper-assertions, are asserted.
- The preservation of the loop invariants and the side conditions of using either a user-specified or automatically selected loop rule are checked modularly via separate Viper methods.
- After the loop, the set of program states is havoced (i.e. the set of states becomes arbitrary), and the loop invariants are assumed. To preserve information about program states from before the loop, overapproximation framing (i.e. Lemma 1 of our paper) is emitted.
- After the loop, depending on the loop rule used, Hypra either assumes that every state satisfies the negated loop guard, or selects those states where the negated loop guard holds
- For modularity reasons, if the loop invariants were to talk about erroneous states, they must only talk about the erroneous states yielded by the loop body but not any erroneous states collected before the loop. In this way, after the loop, the set of erroneous states produced by the loop, as described by the loop invariants, can be easily added to the set of collected erroneous states via a set union operation.

Hypra supports 4 loop rules as shown in Figure 7 of our paper. To specify a loop rule for Hypra to use, provide one of the following 4 keywords following the `while` keyword: `syncRule` (corresponds to WhileSync rule), `syncTotRule` (corresponds to WhileSyncTerm rule), `forAllExistsRule` (corresponds to While-$\forall^*\exists^*$ rule), `existsRule` (corresponds to While-$\exists$ rule). When the loop rule is not specified, it is mandatory to run Hypra with the option `--auto` to enable the automatic selection of loop rules. Figure 8 of our paper shows how Hypra decides which loop rule to use. The same figure can also serve as a guidance on how users should select a loop rule.

All loop rule come with side-conditions:

- `syncRule`: the loop guard must evaluate to the same value in all states.
- `syncTotRule`: the loop guard must evaluate to the same value in all states, and a `decreases` clause (see below) must be provided to prove termination. In addition, if the loop body contains another loop, the nested loop must have a `decreases` clause as well.
- `forAllExistsRule`: in every loop invariant, there must not exist a `forall` quantifier that quantifies over program states after any `exists` quantifier.
- `existsRule`: at least one of the loop invariants must contain a top-level `exists` quantifier that quantifies over program states, and a `decreases` clause (see below) must be provided to prove termination.

The `decreases` clause is useful for proving the termination of a loop. The arithmetic expression a in the `decreases` clause, which is essentially a loop variant, is expected to be **non-negative** in all program states. Its value in a state must **strictly decrease** after one iteration of the loop. Note that the `decreases` clause is simply **ignored** when the rule `syncRule` or `forAllExistsRule` is used, since these rules do not require termination (and termination is not used elsewhere).

Method calls

To call a method with n parameters and m return values, provide exactly n program variables as arguments and exactly m program variables to store the return values. A program variable cannot simultaneously be used as an argument and store the return value in a method call.

.

```
m1() // Method m1 takes 0 parameter and returns 0 value
v1, v2 := m2(v3) // Method m2 takes 1 parameter and returns 2 values
```

Hypra handles method calls modularly. Before a method call, the preconditions of the callee are asserted. After a method call, the set of program states is havoced (i.e. the set of states becomes arbitrary), and the postconditions of the callee are assumed. To preserve information about program states from before the method call, overapproximation framing (i.e. Lemma 1 of our paper) is emitted. In addition, the set of erroneous states produced by the callee, as described by its postconditions, are added to the set of erroneous states of the caller via a set union operation

Running Hypra with custom settings

To verify your own program with Hypra, run Hypra with the command `hypra <path_to_program> <option>*`. You may customize Hypra by providing one or more of the options below as part of the command:

- `--forall`: Only generates overapproximation encodings
- `--exists`: Only generates underapproximation encodings
- `--output <path_to_file>`: Saves the generated Viper program to the specified file
- `--noframe`: Turns off overapproximation framing after loops and method calls
- `--existsframe`: Turns on underapproximation framing after loops
- `--inline`: Verifies the loop invariants in an inline fashion (At the moment, this option does not work well when the `--auto` option is also selected)
- `--auto`: Automatically selects a loop rule to verify loops when users do not specify the rules