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-∀*∃* rule), `existsRule` (corresponds to While-∃ 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