Project Report 1 for 02245

Group 7

02245 Program Verification, Fall Semester 2024

Group Composition

Student Name	**Student ID**	**Contribution**
Oliver Stenberg Jakobsen	s233465	$33.\overline{3}\%$
Federico Marra	s243138	$33.\overline{3}\%$
Amirhosein Rajabi	s243531	$33.\overline{3}\%$

Claimed Features

Feature	Formalized	Implemented	Tested	Stars
Core A	✓	✓	✓	2/2
Core B	✓	✓	✓	2/2
Core C	✓	✓	✓	2/2
Extension 1	✓	✓	✓	1/1
Extension 2	✓	✓	✓	1/1
Extension 3	✓	✓	✓	1/1
Extension 4	✓	✓	✓	2/2
Extension 5	✓	✓	✓	2/2
Extension 6	✓	✓	✓	3/3
Extension 7	✓	✓	✓	1/1
Extension 8	✓	✓	✓	2/2
Extension 9	✓	✓	✓	2/2
Extension 10	✓	✓	✓	2/2
Extension 11	✓	✓	✓	4/4
Total stars				27/27

Contents

Core Feature A: single loop-free method (2⋆)

In this section, we only discuss the semantics, logic, and automation of `match` statements. Other program constructs are discussed in later sections.

Our implementation first translates commands directly into a sequence of four IVL1 commands, i.e. Assert, Assume, Non-deterministic Choice, and Assignments. Then, we transform the resulting IVL1 command to DSA and we end up with a sequence of three IVL0 commands: Assert, Assume, and Non-deterministic Choice. Finally, we compute the weakest precondition sets to compute the weakest preconditions to be checked for satisfiability with Z3. (It is worth mentioning that during the weakest precondition set computation, we insert an Assume right after every Assert for error reporting purposes.) Since we are using the same recursive function as in the lectures to compute the weakest precondition sets, we do not repeat them in the report and just describe the IVL1 encoding for automation. We verify each weakest precondition in its own scope.

Match statements

Operational Semantics We have the following single-step semantic rules for each of the conditions of a `match` statement:

$$\frac{\forall j < i \;\; \mathfrak{m} \not\models b_j \qquad \mathfrak{m} \models b_i}{\langle \text{match}\{b_1 \Rightarrow C_1, \cdots, b_n \Rightarrow C_n\}, \mathfrak{m} \rangle \;\Rightarrow\; \langle C_i, \mathfrak{m} \rangle} \text{cond-}i$$

We also have the following rule if no matching is possible:

$$\frac{\forall 1 \leq i \leq n \;\; \mathfrak{m} \not\models b_i \qquad \forall 1 \leq i \leq n \;\; \mathfrak{m} \not\models b_i}{\langle \text{match}\{b_1 \Rightarrow C_1, \cdots, b_n \Rightarrow C_n\}, \mathfrak{m} \rangle \;\Rightarrow\; \langle \textbf{done}, \mathfrak{m} \rangle} \text{no-match}$$

Program Logic Define $B_i := b_i \wedge \bigwedge_{1 \leq j < i} \neg b_j$ for $1 \leq i \leq n$. For example, $B_1 = b_1$ and $B_2 = b_2 \wedge \neg b_1$. Let $B_{n+1} = \bigwedge_{1 \leq j \leq n} \neg b_j$. Now the following rule gives us the axiomatic semantic counterpart of our operational semantics.

$$\frac{F \wedge B_{n+1} \models G \qquad \forall 1 \leq i \leq n \;\; \vdash \{\!\{\ F \wedge B_i\ \}\!\}\ C_i\ \{\!\{\ G\ \}\!\}}{\vdash \{\!\{\ F\ \}\!\}\ \text{match}\{b_1 \Rightarrow C_1, \cdots, b_n \Rightarrow C_n\}\ \{\!\{\ G\ \}\!\}} \text{match}$$

To make it closer to our automation style and implementation, we can write the proof rule recursively as below:

$$\frac{}{\vdash \{\!\{\ F\ \}\!\}\ \text{match}\{\}\ \{\!\{\ F\ \}\!\}} \text{match-base}$$

$$\frac{\vdash \{\!\{\ F \wedge b_1\ \}\!\}\ C_1\ \{\!\{\ G\ \}\!\} \qquad \vdash \{\!\{\ F \wedge \neg b_1\ \}\!\}\ \text{match}\{b_2 \Rightarrow C_2, \cdots, b_n \Rightarrow C_n\}\ \{\!\{\ G\ \}\!\}}{\vdash \{\!\{\ F\ \}\!\}\ \text{match}\{b_1 \Rightarrow C_1, b_2 \Rightarrow C_2, \cdots, b_n \Rightarrow C_n\}\ \{\!\{\ G\ \}\!\}} \text{match-rec}$$

Automation We denote IVL1 translation of a statement C by $encode(C)$. With this, we encode `match` as follows:

```
[[ match { $b_1 \Rightarrow C_1$, ..., $b_n \Rightarrow C_n$ } ]] :=
{
    assume $b_1$;
    encode($C_1$)
} [] {
    assume $\neg b_1$;
    encode(match { $b_2 \Rightarrow C_2$, ..., $b_n \Rightarrow C_n$ })
}
```

and `[[ match { } ]]` is just Nop operation (or an assert true). For example, the expanded encoding is as follows:

```
{
    assume $b_1$;
    encode($C_1$);
} [] {
    assume $\neg b_1$;
    {
        assume $b_2$;
        encode($C_2$);
    } [] {
        assume $\neg b_2$;
        ...
            {
                assume $b_n$;
                encode($C_n$);
            } [] {
                assume $\neg b_n$
            }
    }
}
```

Variable declarations

We have a method `get_fresh_var_name(ident: Ident)` that given an identifier, returns a variable name of the form `{ident}_{uuid}` guaranteed to be fresh. Whenever we encounter a variable declaration, we generate a fresh variable with this method and assign it to the declared variable.

Verification of methods

To verify a method, we need to assume preconditions F and assert postconditions G at its exit points (end of method or returns).

For preconditions, we directly assert them at the beginning of the method, because

$$wps(\texttt{assume F; body})[G] = \{F \longrightarrow w \ : \ w \in wps(\texttt{body})[G]\}$$

and we assert $\neg w$ for each $w \in wps(\texttt{body})[G]$ with Z3 and expect it to be unsatisfiable to prove validity of $F \longrightarrow w$. Notice, $\neg(F \longrightarrow w) \equiv \neg(\neg F \vee w) \equiv F \wedge \neg w$. Therefore, we can assert the precondition F and negation of each weakest precondition of body.

Normally, one would compute the weakest preconditions of the body of the method beginning with postconditions. However, we start the weakest precondition with an empty set of postconditions. Namely, we prove the safety of methods. To check postconditions of methods, we only assert them when they return. To support void methods without return statements, we artificially insert a `return result;` at the end of such methods. Notice that `result` can only appear in postcondition; therefore, it will be distinguished with other `return` statements in the body. However, for methods that are expected to return a value, we insert an `assert false` with the message "Method might not return" to warn about methods that might not return. The underlying reason is that all possible execution paths of such methods should end with a return statement and if the execution falls off the end of a method without encountering a return, it means the method might not return. See section on return statements for encoding of return statements.

Passing example: The following example verifies according to our formalization and implementation:

```
method m(x: Int) {
    var y: Int := x + 15;
    match {
      x > 0 =>
        assert y/x > 0,
      true =>
        y := 5
    };
    assert y > 0
}
```

Failing example: The following example leads to a verification failure according to our formalization and implementation:

```
method m(x: Int): Int
    requires x >= 0
    ensures result >= 0
    // false: (consider execution when x = 1)
    // @CheckError
    ensures result == 3 * x
{
    var acc: Int := 0;
    acc := x / 2;
    acc := acc * 6;

    return acc
}
```

Comments:

- We check the well-definedness of expressions by only checking **divisions by zeros**. We only check division by zeros in variable declarations, assignments, conditions, returns, methods, and function calls by adding division by zero assertions right before such expressions to keep context. However, we do not check them in assertions or assumptions as logical formulae may create a context that division by zero is meaningless. For example $\forall x(x/x == 1)$.

- test/division-by zero-tenary-good.slang

  ```
  method client() {
      var x : Int;
      x := true ? 1 : 1 / 0
  }
  ```

- test/division-by-zero-tenary-bad.slang

  ```
  method client() {
      var x : Int;
      // @CheckError
      x := true ? 1 / 0 : 1
  }
  ```

- test/division-by-zero-double.slang

  ```
  method client() {
      var x: Int;
      // @CheckError
      x := x / ((2 * x / x) - 1);
      // @CheckError
      x := x / ((x / x) - 1)
  }
  ```

- We do not allow shadowing or redeclarations of variables. Namely, a variable with a name existing in the current method can not be declared again inside a match or loop. This means that we do not begin a new scope with a loop or match. We check for shadowed variables with a DFS before translation of programs to IVL. If the purpose of the redeclaration is havocing, one can naturally proceed as follows:

  ```
  Var x := a;
  ...
  Var x; // wrong
  Var y;
  x := y; // correct
  ```

Core Feature B: partial correctness of loops (2$\star$)

We support verification of partial correctness of unbounded `loop` commands.

Operational Semantics We define a partially correct loop as a loop that does not check for termination and therefore looks like this:

$$\text{loop}\{b_1 \Rightarrow C_1, b_2 \Rightarrow C_2, ..., b_n \Rightarrow C_n\}$$

This is interpreted as if b_1 holds than C_1 is executed. If b_1 does not hold then b_2 is checked to see if C_2 should execute (if a second case exists). It is important to note that a loop must have at least one of these cases but not necessarily more than that. Once no case holds the loop exists.
The operational semantics for this is:

$$\frac{\forall j < i \;\; \mathfrak{m} \not\models b_j \qquad \mathfrak{m} \models b_i \qquad \langle\, C_i, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \textbf{done}, \mathfrak{m}' \,\rangle}{\langle\, \text{loop}\{b_1 \Rightarrow C_1, ..., b_n \Rightarrow C_n\}, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, C_i; \text{loop}\{b_1 \Rightarrow C_1, ..., b_n \Rightarrow C_n\}, \mathfrak{m} \,\rangle}\;\text{loop-repeat}$$

$$\frac{\forall i \leq n \;\; \mathfrak{m} \not\models b_j}{\langle\, \text{loop}\{b_1 \Rightarrow C_1, ..., b_n \Rightarrow C_n\}, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \textbf{done}, \mathfrak{m} \,\rangle}\;\text{loop-stop}$$

Program Logic We introduce a new inference rule for the loop command:

$$\frac{\vdash \{\{\; I \;\}\}\; \text{match}\{b_1 \Rightarrow C_1, b_2 \Rightarrow C_2, ..., b_n \Rightarrow C_n\}\; \{\{\; I \;\}\}}{\vdash \{\{\; I \;\}\}\; \text{loop}\{b_1 \Rightarrow C_1, b_2 \Rightarrow C_2, ..., b_n \Rightarrow C_n\}\; \{\{\; I \wedge \bigwedge_{i \leq n} \neg b_i \;\}\}}\;\text{loop}$$

We now want to prove the triple:

$$\{\{\; I \;\}\}\; \text{loop}\{b_1 \Rightarrow C_1, b_2 \Rightarrow C_2, ..., b_n \Rightarrow C_n\}\; \{\{\; I \wedge \bigwedge_{i \leq n} \neg b_i \;\}\}$$

In this triple I must hold once the loop terminates and the loop should only terminate if all of the conditions are false. To make sure I holds after the loop we have the premise that if a branch is taken then after executing the command of that branch then I must hold. This is only a requirement for reachable branches which is why we in loop-repeat have a premise saying that we must enter the first possible branch whose condition is true. This ensures that I holds after the loop because no matter which path we take through the loop it will hold. Finally we show that all the conditions for the branches of the loop must be false once the loop terminates due to loop-stop which requires them to be false in the premise.

Automation To automate the verification we encode the command

$$\text{loop}\{b_1 \Rightarrow C_1, b_2 \Rightarrow C_2, ..., b_n \Rightarrow C_n\}$$

To encode this into IVL1 we need to find the variables that are modified in the loop. These variables we store in $\overline{x}$. To encode into IVL1 we follow these steps:

1. Assert the invariants to make sure they hold on entry.
2. Remove concrete values from $\overline{x}$ by havocing and then assume the invariants to potentially limit the values for $\overline{x}$.

3. For each branch we introduce a non-deterministic choice. The first choice assumes the condition and asserts the invariants after executing the command. The other branch assumes the condition does not hold and then moves on to the next branch. This way we get nested non-deterministic choices.

Abstract encoding would be:

```
assert $I$;
havoc $\overline{x}$;
assume $I$;
encode(match {
    $b_1 \Rightarrow C_1$; assert $I$; assume false,
    ...,
    $b_n \Rightarrow C_n$; assert $I$; assume false
})
```

The expanded encoding is as follows in IVL1:

```
assert $I$;
var $\overline{x_{new}}$: $\overline{T_x}$;
$\overline{x}$ := $\overline{x_{new}}$;
assume $I$;
{
    assume $b_1$;
    encode($C_1$);
    assert $I$;
    assume false
} [] {
    assume $\neg b_1$;
    {
        assume $b_2$;
        encode($C_2$);
        assert $I$;
        assume false
    } [] {
        assume $\neg b_2$;
        ...
            {
                assume $b_n$;
                encode($C_n$);
                assert $I$;
                assume false
            } [] {
                assume $\neg b_n$
            }
    }
}
```

Passing example:

```
method sumn(n: Int): Int
    requires n >= 0
    ensures result == n * (n + 1) / 2
{
    var acc: Int := 0;
    var i: Int := 0;
    loop
        invariant i >= 0
        invariant i <= n
        invariant acc == i * (i + 1) / 2
    {
        i < n ⇒
            i := i + 1;
            acc := acc + i
    };
    assert i == n;
    assert acc == n * (n + 1) / 2;

    return acc
}
```

Failure example:

```
method iter(n: Int): Int
    requires n >= 0
{
    var i: Int := 0;
    loop
        // false:  when the loop terminates, i == n + 1
        invariant i <= n
    {
        i <= n ⇒ i := i + 1
    }
}
```

Core Feature C: error reporting (2$\star$)

We have used the infrastructure provided to perform error reporting. To get the red line placement in the correct place we have chosen to use the weakest precondition set method. This also allows us to mask errors when appropriate. A part of this masking is that we do not report errors for code branches that are unreachable. As part of error reporting we also have a message. In the following descriptions of the message text we use the notation [...] which should be read as a string interpolation where the value of ... is inserted into the string and the [] are removed. We have error reporting for the following behavior.

Assertion failure

When an assertion might fail we underline the expression and provide the message.

"Assertion might fail"

Division by zero

The way we check for division by zero is by extracting all divisors in an expression and then adding an assertion like this where e_{div} is the divisor:

$$\text{assert } e_{div} == 0$$

Doing it this way allows us to handle case where the divisor is an expression and not simply a variable. We have division by zero in the following places:

- Variable assignment
- Conditions for match
- Conditions for loops
- Conditions for unbounded for loops
- Arguments passed to method
- Arguments passed to function
- Return statements

An important note here is that modulo can also lead to a division by zero error so when checking in above cases we treat module as division. The reason we do not have it for assertions is that this automatically checked by z3. If a potential division by zero is found the following message is given.

"Possible division by zero!"

Loop invariants

Loop invariants have to hold on loop entry and they must also hold after each loop iteration. Since these are two different conditions we have given them different error messages. For both types of errors the red line is drawn underneath the invariant itself.

The loop entry we simply have the message:

"Loop invariant [INVARIANT EXPRESSION] might not hold on entry"

The preservation of invariants is a bit more complex because a loop can consist of multiple branches. To make it easier to detect why the invariant is failing we have therefore chosen to include the branch condition in the error message to make it easier to identify the conflicting code. The message is therefore:

"Loop invariant [INVARIANT EXPRESSION] might not be preserved in case [BRANCH CONDITION]"

Another way an invariant might not be preserved is when keywords like `continue` or `break` is used. In case the invariants are not satisfied we underline the `continue` or `break` which violates the invariant and provide the following messages respectively:

"Invariant [INVARIANT EXPRESSION] might not hold after continue"

"Invariant [INVARIANT EXPRESSION] might not hold after break"

Loop variants

Loop variants also have two different ways of breaking namely that they must be non-negative on entry and they must decrease after each loop iteration. This decrement must happen no matter which branch is taken in the loop body and can be used to determine if a loop terminates. This is because the loop should not continue once the variant goes negative and since it is determined to be non-negative on entry the loop can only run a finite amount of times.

When a variant does not hold on entry we give the following message underneath the variant itself:

"Loop variant might not be non-negative on entry"

As mentioned the variant must decrease no matter which branch is entered. To make error finding easier we have therefore chosen to put the red line under the branch condition in case the body of that branch does not decrease the variant. The message provided is:

"Loop variant might not be decreased in case [BRANCH CONDITION]"

Here there is also a special case for `continue`. In case the variant has not decreased before a `continue` is encountered then it is underline with a red and give the following message:

"Loop variant might not be decreased before continue"

Method not returning

When a method is declared the specification also describes wether the method should return something or not. When a method should return we therefore need to make sure that no matter which path is taken it returns. In the case it does not return we provide the following error message with a red line underneath the method name:

"Method might not return"

Method post-conditions

Post-conditions of a method must hold once the method returns no matter if there is a value associated with that return. This is because post-conditions might contain global variables. To check for this we therefore pass the post-conditions along when we are translating to IVL because that way we can insert assertions when a `return` is encountered. In case a post-condition for a method is not satisfied we put the red line under the return and then we provide the following message where the post-condition is the first one that the does not hold (this is in case a single return violates multiple post-conditions):

"Ensure [POST-CONDITION] might not hold"

Method variant

Method variants are used to determine if a recursive method will terminate. In our implementation we can have methods that recursively call each other but we only check the variant when the method calls itself. The way this is checked is by calculating the value of the variant when the method is entered when the method then calls itself we use the provided arguments to calculate what the variant would be for that method call. If the latter variant is not strictly smaller than the calculation performed at method entrance then we place the red line under the line which has the self-recursing method call and provide the following message:

Method variant might not be decreased

Method calls satisfying pre-conditions

When a method is called the body of the method is not used for verification, instead we rely only on the pre- and post-conditions. It is therefore important to check the precondtions of a method when it is called because they must be satisfied in order for the post-conditions to be used in future verification. When we see a method call we therefore check if the arguments satisfy the method's pre-conditions (along with potential global variable state). If this is not the case then we put a red line underneath the line where the method call happens and we provide the following message. Where we refer to the pre-condition which is not satisfied:

Required [PRE-CONDTION] might not hold

Method has wrong modifies

When a method modifies a global variable either directly or indirectly through a call to another method it must mention the variable name it its **modifies** clause. If it does not we underline the variable assignment of the global variable and supply the following message:

Variable [GLOBAL VARIABLE NAME] is modified but not declared in the current scope. If it is a global variable, consider annotating the method.

Extension Feature 1: bounded for-loops (1⋆)

We have interpreted a bounded loop as something where the range of the iterator only consists of two numbers, so both the start and the end is a literal. In the body of `for` loops we do not allow for modification of the iterator variable which means that if the body tries to modify the variable then the program is invalid. This means that we do not consider a loop with the following range as bounded:

```
for i in 0..(5 + 5) {
    ...
}
```

Operational Semantics A valid bounded for-loop takes the following form in which l refers to a literal number.

$$\text{for } x \text{ in } l_1 \text{ .. } l_2 \ \{ \ C \ \}$$

The operational semantics of a bounded for-loop can be translated to repeating the body x times, corresponding to the difference between the start and end in the interval. We have chosen that our for-loops are inclusive for the start but exclusive for the end. The operational semantics is therefore:

$$\frac{l_1 < l_2}{\langle \text{for } x \text{ in } l_1 \text{ .. } l_2 \ \{ \ C \ \}, \mathfrak{m} \rangle \ \Rightarrow \ \langle C; \text{for } x \text{ in } l_1 + 1 \text{ .. } l_2 \ \{ \ C \ \}, \mathfrak{m}[x \leftarrow l_1] \rangle} \text{ for-bounded}$$

$$\frac{l_1 \geq l_2}{\langle \text{for } x \text{ in } l_1 \text{ .. } l_2 \ \{ \ C \ \}, \mathfrak{m} \rangle \ \Rightarrow \ \langle \textbf{done}, \mathfrak{m} \rangle} \text{ for-bounded-end}$$

Program Logic Since a bounded for loop is just unfolded it is the same as a sequence of the same command repeated $l_2 - l_1$ times. This can be zero times if $l_1 \geq l_2$. The Hoare rules needed for bounded for loops is as follows:

$$\frac{l_1 \geq l_2}{\vdash \{\!\{ \ F \ \}\!\} \text{ for } x \text{ in } l_1 \text{ .. } l_2 \ \{ \ C \ \} \ \{\!\{ \ F \ \}\!\}} \text{ for-bounded-skip}$$

$$\frac{\vdash \{\!\{ \ F \ \}\!\} \ x := l_1; C; x := l_1 + 1; C; ...; x := l_2 - 1; C; x := l_2 \ \{\!\{ \ G \ \}\!\}}{\vdash \{\!\{ \ F \ \}\!\} \text{ for } x \text{ in } l_1 \text{ .. } l_2 \ \{ \ C \ \} \ \{\!\{ \ G \ \}\!\}} \text{ for-bounded}$$

We now want to prove the triple:

$$\{\!\{ \ F \ \}\!\} \text{ for } x \text{ in } l_1 \text{ .. } l_2 \ \{ \ C \ \} \ \{\!\{ \ G \ \}\!\}$$

There are two cases for bounded for-loops; the loop never runs or the loop runs once or more times. When the loop never runs the triple is simple to prove because then we must simply have that $F \models G$. When the loop runs more than once we have that x is assigned a value equal to the start value plus the number of iterations done already. After having done this enough times we finally assign l_2 to x after which G must hold. This can simply be proved using the Hoare rules for sequence.

Automation It is quite simple to automate the verification of bounded for-loops because we can just unfold it. In case the loop body should not be executed ($l_1 \geq l_2$) it is just translated to an assertion of **true**. In the other case we just repeat the command along with assignments of the iterator. If the program has an assertion after the loop it will simply be prefixed by all of executions of the command like in the Hoare rule for-bounded. In IVL1 it looks like this:

```
x := $l_1$;
encode($C$);
x := $l_1$ + 1;
encode($C$);
...
x := $l_2$ - 1;
encode($C$);
x := $l_2$
```

Passing example: Example with positive range

```
method client() {
    var acc: Int := 0;
    for i in 0..10 {
        acc := acc + i
    };
    assert acc == 9 * 10 / 2
}
```

Passing example: Example with negative range

```
method client() {
    var acc: Int := 0;
    for i in -5..2 {
        acc := acc + i
    };
    assert acc == -14
}
```

Failing example: Fails because acc is actually 4 and not 5.

```
method client() {
    var acc1: Int := 0;
    for i in -1..5 {
        acc := i
    };
    // Fails here
    assert acc == 5
}
```

Extension Feature 2: verification of mutually recursive methods (1⋆)

We implemented and tested the support for partial correctness verification of multiple recursive methods. We also support possibly mutually recursive methods but we will discuss this topic in the section extension 7.

Operational Semantics We define the main idea for the operational semantics of partial correctness of methods into two parts, the semantics for the call of the method and for the declaration.

The semantic of the **declaration** is:

```
method name( x : T ): S  // method name with I/O  (only one)
    requires F           // pre-condition         (zero or more)
    ensures G            // post-condition        (zero or more)
    { C }                // body                  (zero or one)
```

- **method name**: here we define the input variable vector with their type and the output type of the method;
- **requires**: it stands for the pre-condition of the whole method and can only contain input variables;
- **ensures**: it stands for the post-condition of the whole method and can contain input and output variables;
- **body**: {C} stands for the command executed when the method is called, if there is not a body, the method is abstract.

The semantic of a method call is:

$$\overline{z} := \text{foo}(\overline{e})$$

So we can compose the proof tree for the method call like this:

$$\frac{}{\langle\, \overline{z} := \text{foo}(\overline{e}), \mathfrak{m}\,\rangle => \langle\, \text{inline}[[\overline{z} := \text{foo}(\overline{e})]], \mathfrak{m}\,\rangle}\ \text{method call with assignment}$$

The main idea for method calls perfectly refers to the slides [06-method.pdf:11/44]. We will talk about **return** in the section extension 10.

Program Logic We make an essential assumption: the variables in $\overline{z}$ are not allowed to be arguments $\overline{e}$.

$$\frac{\{\{\ F\ \}\}\ \textbf{method}\ \text{foo}(\overline{x})\ \{\{\ G\ \}\}}{\vdash \{\{\ F[\overline{x} := \overline{e}]\ \}\}\ \overline{z} := \text{foo}(\overline{e})\ \{\{\ G[\overline{x, y} := \overline{e, z}]\ \}\}}\ \text{method call (w/o framing rule)}$$

Applying the framing rule that we can get from the slides to the method call program logic, assuming that $\overline{z}$ contains no free variable also contained into H, we can obtain this proof tree:

$$\frac{\{\{\ F\ \}\}\ \textbf{method}\ \text{foo}(\overline{x})\ \{\{\ G\ \}\}}{\vdash \{\{\ F[\overline{x} := \overline{e}] \wedge H\ \}\}\ \overline{z} := \text{foo}(\overline{e})\ \{\{\ G[\overline{x, y} := \overline{e, z}] \wedge H\ \}\}}\ \text{method call w framing rule}$$

Automation For the automation of the method call:

$$\overline{z} = \text{method}(x);$$

we implemented using IVL1, translating this:

```
assert F;   // requires
havoc $\overline{z}$;    // var to which is assigned to the return value
assume G    // ensures
```

into this following IVL1 automation, where we substitute the havoc with an assignment to a new fresh variable (just declared).

```
assert F;
var $\overline{new\_var}$: z_type; // declaration of fresh var
$\overline{z}$ := $\overline{new\_var}$; // assignment of the fresh var to the one to havoc
assume G
```

For the method declaration itself we have the check executed just by slang language syntax.

Passing example:

```
method sumn(n: Int): Int
    requires n >= 0
    ensures result == n * (n + 1)/2
{
    match {
        n == 0 =>
            return 0,
        true =>
            var res: Int;
            res := sumn(n - 1);
            return res + n
    }
}
```

Failing example: An example where is detected an error in the `requires`

```
method sum(n: Int) : Int
    // @CheckError
    requires n > 0
    ensures result == 9
{
    var y: Int := 7;
    return y + 2
}

method client() {
    var z: Int;
    z := sum(-1);
    assert z == 9
}
```

In this other example the error is in the `ensures`

```
method Sum(n: Int) : Int
    requires n >= 0
    // @CheckError
    ensures result == (n == 0 ? -1 : n + (n - 1) * (n - 1))
{
    var sq: Int;
    sq := Square(n - 1);
    match {
        n == 0 => return -1,
        true => return n + sq
    }
}
method Square(x: Int) : Int
    ensures result == 2 * x
{
    return 2 * x
}
```

Extension Feature 3: efficient assignments (DSA) (1⋆)

We implemented the efficient verification of assignments by transforming every command into dynamic single assignments (DSA) before computing weakest preconditions,finishing with a subset of `IVL1` comprised only of `Assert`, `Assume`, `Seq` and `NonDet`.

Operational Semantics

$$\frac{}{\langle\, x := e, \mathfrak{m} \,\rangle \Rightarrow \langle\, \textbf{done}, \mathfrak{m}[x \leftarrow [e](\mathfrak{m})] \,\rangle}\ \text{assignment}$$

Automation To automate the verification we encode the command

$$\text{Var } x : \text{T}; x := e_1; x := x + e_2$$

For the automation in our project we instantiated an Hash Map, in which we save the latest value of every variable (i.e. x maps to x_{1000} if x_{1000} is its latest version), when we make a new assignment to the variable, we update that value, or if it is inside an expression, we recall the value from the Hash Map.

```
Var x: T;
x1 := e1;
assume x1 == e1;
x2 := x1 + e2;
assume x2 == x1 + e2
```

If we have a non deterministic choice we copy the Hash Map for each branch and we update or get the value of the variables just from the Hash Map corresponding to that branch. To synchronize branches, we create a new fresh variable x' for every variable x and assign the last version of x in each branch to x'. If a value appears only in one branch, it will be havoced in the branch it does not appear.

```
Var x: T;
x1 := e1;
{
    x2 := x1 + e2;
    assume x2==x1+e2
} [] {
    x3 := x1 + e3;
    assume x3==x1+e3
}
```

Passing example

```
method main () {
    var x : Int ;
    var y : Int ;
    var z : Int ;
    assume x >= 0 && y >= 0;
    z := x - y ;
    match {
        z < 0 =>
            z := z + y ;
            z := z + 2 * x,
        true =>
            z := z - x ;
            z := z + 4 * y,
    };
    assert z <= 3 * x && z <= 3 * y && ( z == 3 * x || z == 3 * y )
}
```

Rejecting example This example passes because it it detected an error under the line with `@CheckError`. It is the post-condition of not the latest version of z, that is why it gives error at that line and not at the one below (the last one).

```
method main () {
    var x : Int ;
    var y : Int ;
    var z : Int ;
    assume x >= 0 && y >= 0;
    z := x - y ;
    match {
        z < 0 =>
            z := z + y ;
            z := z + 2 * x,
        true =>
            z := z - x ;
            z := z + 4 * y,
    };
    // @CheckError
    assert (z < 0 && z == x - y + 2 * x) || (z >= 0 && z == x - y + 4 * y);
    assert (z < 0 && z == x - y + y + 2 * x) || (z >= 0 && z == x - y - x + 4 * y)
}
```

Extension Feature 4: unbounded for-loops (2⋆)

In our definition for bounded loops we only allowed for number literals so if the range of `for` command contains either variables or arithmetic operators we consider it an unbounded loop. We have chosen to encode these unbounded `for` loops using the `loop` command. This is done by turning the `for` into a `loop` with a single branch where the condition is the range of the `for` loop and the command for the branch is the body of the `for`. On top of this we have invariants describing that the iterator variable stays within the range where this comparison is inclusive in both ends. Just like with bounded `for` loops we do not allow for modification of the iterator variable which means that if the body tries to modify the variable then the program is invalid. As part of this encoding we also increase the iterator by one at the end of the body.

Operational Semantics We already have operation semantics for handling a loop so here we just need one rule where we translate `for` into `loop`. An important note here is that x is assigned to the value of e_1 before the `loop` execution begins.

$$\frac{e_1 \text{ is not a literal} \vee e_2 \text{ is not a literal}}{\langle \text{for } x \text{ in } e_1..e_2\{\ C\ \}, \mathtt{m}\,\rangle \Rightarrow \langle\, x := e_1; \text{loop}\{(x \geq e_1 \wedge x \leq e_2) \Rightarrow (C; x := x + 1)\}, \mathtt{m}\,\rangle}\ \text{unbounded-for}$$

Program Logic Since the loop is unbounded we are dealing with some code which might not terminate so we will be dealing with invariants instead of just normal pre- and post-conditions. These conditions will of course be able to reason about using the same framing rule that has been mentioned previously. To keep the rule a bit shorter we define the following:

$$x \text{ in range} = (x \geq e_1 \wedge x \leq e_2)$$

The program logic for unbounded `for` is therefore:

$$\frac{\vdash \{\{\ I \wedge x := e_1 \wedge x \text{ in range}\ \}\}\ \text{loop}\{(x \geq e_1 \wedge x \leq e_2) \Rightarrow (C; x := x + 1)\}\ \{\{\ I \wedge x \text{ in range}\ \}\}}{\vdash \{\{\ I\ \}\}\ \text{for } x \text{ in } e_1..e_2\{\ C\ \}\ \{\{\ I \wedge \neg(x \geq e_1 \wedge x \leq e_2)\ \}\}}\ \text{unbound}$$

Automation In the automation of this we also used the translation into `loop` as a stepping stone so abstractly the encoding is:

x `:=` e_1`;`
`encode(loop {` $(x \geq e_1 \wedge x \leq e_2) \Rightarrow (C; x := x + 1)$ `})`

Which when translated to IVL1 becomes:

```
$x$ := $e_1$;
assert $I$;
var $\overline{m_{new}}$: $\overline{T_m}$;
$\overline{m}$ := $\overline{m_{new}}$;
assume $I$;
{
  assume $(x \geq e_1 \wedge x \leq e_2)$;
  encode($C$);
  $x$ := $x + 1$;
  assert $I$;
  assume false
}
[]
{
  assume $\neg(x \geq e_1 \wedge x \leq e_2)$
}
```

Passing example:

```
method client(n: Int): Int
    requires n >= 0
    ensures result == n * (n + 1) / 2
{
    var acc: Int := 0;
    for i in 1..(n + 1)
        invariant acc == i * (i - 1) / 2
    {
        acc := acc + i
    };
    return acc
}
```

Failing example: This test is failing because when i reaches 10 then the counter resets. Since there is no upper bound for n this can happen in some cases.

```
method client(n: Int)
    requires n >= 0
{
    var counter: Int := 0;
    for i in 0..n
        // Fails here
        invariant counter == i
    {
        counter := counter + 1;
        match {
            i == 10 => counter := 0
        }
    };
    assert counter == n
}
```

Extension Feature 5: custom type definitions (2⋆)

We support verification of *slang* programs that involve the declaration, axiomatisation, and use of custom types in the form of domains.

Operational Semantics There is no executable code for this feature so no operational semantics is defined.

Program Logic We only assert the axioms directly in Z3. If there is a model that satisfies our axiom, our verification would be sound. Otherwise, unsatisfiable results can not be interpreted as validity because there is no model even before asserting the negation of the weakest preconditions.

Automation We translate each axiom as an assertion directly in SMT and each function as a function declaration. There is no IVL translation for this feature. Sort is defined automatically by slang library.

Passing example:

```
domain Pair {
  function pair(x: Int, y: Int): Pair
  function fst(x: Pair): Int
  function snd(x: Pair): Int

  axiom forall x: Int :: forall y: Int :: fst(pair(x,y)) == x
  axiom forall x: Int :: forall y: Int :: snd(pair(x,y)) == y
}

method client(x: Int, y: Int)
{
  var xy: Pair := pair(x,y);

  assert fst(xy) == x;
  assert snd(xy) == y
}
```

Failing example:

```
domain X {
    function X(n: Int, m: Int): Int
    axiom
        forall n: Int :: forall m: Int ::
            (n >= 0 && m >= 0) ==> X(n, m) == (m == 0 ? 0 : n + X(n, m - 1))
}

method client() {
    assert X(5, 0) == 0;
    assert X(5, 1) == 5;
    // Fails here
    assert X(5, 2) == 15
}
```

Extension Feature 6: user-defined functions (3$\star$)

We support user-defined functions that can be defined by providing the function body, pre- and post- conditions, or both.

We verify user-defined functions in two steps. First, we verify the body of the function complies with the postconditions in a new Z3 scope. We elaborate on it in a few moments. If the verification is successful, we add the function as an axiom illustrated in the lecture notes. Otherwise, we refrain from adding the axiom to the program since verification failure hints at the axiom being unsound and causes unsoundess in the rest of the program.

To verify the body of a function, we sacrifice completeness for soundness. Namely, to prevent adding an unsound axiom to our scope and detect non-compliance of a function postcondition with its body, we add $F \Rightarrow G[\textbf{result} \leftarrow f(x)]$ as an axiom where F is the preconditions and G is the postconditions of function $f(x)$. Next, we translate the function to a method similar to the lectures except for removing the postcondition `ensures result == f(x)`. (because we are removing $f(x) = e$ from the axiom and the solver does not have any clue about the body of f anymore.) This limits us to proofs that need at most 1 unfolding of the body and brings us to the fact that we are less complete.

We are less complete because when verifying the method, the only information we have about $f(x)$ in the axiom is the postconditions of the function. Therefore, we will fail to verify the body of the method if it is needed to unfold the function call in the body of the function to prove the postconditions. It is noteworthy that verifying such properties that need more than one unfolding of the function body could be tricky without proper patterns for quantifier instantiation as they may quickly lead to unknown results and matching loops. Hence, we claim that our choice of sacrifice of completeness for soundness is plausible. It is also worth mentioning that if we manage to verify the method body (or the function body), we add the function as a full axiom (with the function body rather than just the preconditions and postconditions) illustrated in the lectures since we won't face unsoundness with the postconditions anymore.

Functions are translated to methods of the following form during verification of their bodies: (Red lines are not included in our encoding.)

```
method check_f($\overline{x}:\overline{T}$) : S
    requires F
    ensures G
    ensures result == f(x)
{
    return $e$
}
```

Functions are translated to axioms of the following form during verification of their bodies: (Red lines are not included in our encoding.)

```
axiom {
    forall $\overline{x}:\overline{T}$ :: $F \longrightarrow f(x) == e \land G[\texttt{result} \leftarrow f(x)]$
}
```

Remark: We do not check that the preconditions of the function are satisfied in function calls. In other words, we do not check for well-definedness of expressions except for division by zeros. Our implementation does not capture the notion of limited functions.

Passing example:

```
function fac(n: Int): Int
    requires n >= 0
    ensures result > 0
{
    n == 0 ? 1 : n * fac(n - 1)
}

method client() {
    assert fac(0) == 1
}
```

Failing example:

```
function fac(n: Int): Int
    requires n >= 0
    ensures result > 0
{
    n == 0 ? 0 : n * fac(n + 1)
}
```

Extension Feature 7: total correctness for methods ($\star$)

We support total correctness verification of recursive methods.

Operational Semantics We define the main idea of total correctness of methods starting from the semantic of the declaration:

```
    method name( x : T ): S  // method name with I/O
        requires F           // pre-condition
        ensures G            // post-condition
        decreases V          // variant
        { C }                // body
```

- **decreases**: a variant certifies the termination of the method call chain.

For the operational semantics of the method call, we can refer to section extension 2.

Program Logic We extended the program logic present in the section extension 2 by adding the variant V, to this proof tree not considering the framing rule:

$$\frac{\{\!\{\ F\ \}\!\}\ \textbf{method}\ \mathrm{foo}(\overline{x})\ \{\!\{\ G\ \}\!\} \qquad \overline{p}\ \text{fresh variables}}{\vdash \{\!\{\ (F \land V < v)[\overline{x} := \overline{p}] \land \overline{p} == \overline{e}\ \}\!\}\ \overline{z} := \mathrm{foo}(\overline{e})\ \{\!\{\ G[\overline{x,y} := \overline{e,z}]\ \}\!\}}$$

where the fresh variable v saves the initial value of variant V

Automation We extended the automation implementation from the one in section extension 2, adding an assertion to the variant just before the call to the recursive method.

```
v_old := v;
assert v_old ≥0;
assert F;
encode(C);
v_new := v;
assert v_new<v_old;
var new_var: S;
z := new_var;
assume G
```

For the actual implementation of the third line, we did not use a macro but instead when we store locally the value of the variant before the body commands, and we check if the new value of the variant is less than the old one.

Passing example:

```
method sumn(n: Int): Int
    requires n >= 0
    ensures result == n * (n + 1)/2
    decreases n
{
    var ret: Int;
    match {
        n == 0 =>
```

```
            ret := 0,
        true =>
            var res: Int;
            res := sumn(n - 1);
            ret := res + n
    };
    return ret
}
```

Failing example: This example fails because `assert` $v_{new} < v_{old}$ does not hold (assert $n + 1 < n$ fails).

```
method sumn(n: Int): Int
    requires n >= 0
    ensures result == n * (n + 1)/2
    decreases n
{
    var ret: Int;
    match {
        n == 0 =>
            ret := 0,
        true =>
            var res: Int;
            // @CheckError
            res := sumn(n + 1);
            ret := res + n
    };
    return ret
}
```

Extension Feature 8: total correctness of loops (2⋆)

Our program supports the total correctness verification of loops.

Operational Semantics Operational semantics for total correct loops needs to be extended from that in section core B. We add to the invariant a variant V that permits us to check the end of a loop.

We can summarize the loops' syntax into the following:

```
loop
    invariant I
    decreases V
{
    b_1 => C_1
    []
    ...
    []
    b_n => C_n
}
```

We need to extend operational semantics from the one explained in the section core B because the not holding of any b_i is not the only cause of a possible loop-stop.

$$\frac{\forall j < i \;\; \mathfrak{m} \not\models b_j \qquad \mathfrak{m} \models b_i \qquad \mathfrak{m} \models v_p \qquad \langle\, C_i, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \textbf{done}, \mathfrak{m}' \,\rangle}{\langle\, \text{loop}\{b_1 \Rightarrow C_1, ..., b_n \Rightarrow C_n\}, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, C_i; \text{loop}\{b_1 \Rightarrow C_1, ..., b_n \Rightarrow C_n\}, \mathfrak{m} \,\rangle} \text{ loop-repeat}$$

$$\frac{\forall i \leq n \;\; \mathfrak{m} \not\models b_j \qquad \forall p \leq k \;\; \mathfrak{m} \not\models v_p}{\langle\, \text{loop}\{b_1 \Rightarrow C_1, ..., b_n \Rightarrow C_n\}, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \textbf{done}, \mathfrak{m} \,\rangle} \text{ loop-stop at k iteration}$$

$$\frac{\forall i \leq n \;\; \mathfrak{m} \models b_j \qquad \forall p \leq k \;\; \mathfrak{m} \not\models v_p}{\langle\, \text{loop}\{b_1 \Rightarrow C_1, ..., b_n \Rightarrow C_n\}, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \textbf{done}, \mathfrak{m} \,\rangle} \text{ loop-stop at k iteration}$$

$$\frac{\forall i \leq n \;\; \mathfrak{m} \not\models b_j \qquad \forall p \leq k \;\; \mathfrak{m} \models v_p}{\langle\, \text{loop}\{b_1 \Rightarrow C_1, ..., b_n \Rightarrow C_n\}, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \textbf{done}, \mathfrak{m} \,\rangle} \text{ loop-stop at p iteration}$$

We define v_p as the series of values that gets the variant v at iteration p.

Program Logic Like in the section core B we define $B_i := b_i \wedge \bigwedge_{1 \leq j < i} \neg b_j$ for $1 \leq i \leq n$. But we also add v_i that, like in the previous paragraph, is the series of value that the variant gets, from the first iteration to the last one. The following rule gives us the axiomatic semantic counterpart of our operational semantics. In the precondition we refer to the old value of v, before the body of the loop changes it.

$$\frac{\vdash \{\{\; I \wedge v \geq 0 \;\}\} \text{ match}\{b_1 \Rightarrow C_1, b_2 \Rightarrow C_2, ..., b_n \Rightarrow C_n\} \;\{\{\; I \wedge v_{new} < v_{old} \;\}\}}{\vdash \{\{\; I \wedge v \geq 0 \;\}\} \text{ loop}\{b_1 \Rightarrow C_1, b_2 \Rightarrow C_2, ..., b_n \Rightarrow C_n\} \;\{\{\; I \wedge v_{new} < v_{old} \wedge \bigwedge_{i \leq n} \neg b_i \;\}\}} \text{ loop}$$

Automation Abstract encoding would be:

```
$v_{old} := v$;
assert $v_{old} \geq 0$;
assert $I$;
havoc $\overline{x}$;
assume $I$;
encode(match {
    $b_1 \Rightarrow C_1$; assert $I$; $v_{new} := v$; assert $v_{new} < v_{old}$; assume false,
    ...,
    $b_n \Rightarrow C_n$; assert $I$;$v_{new} := v$; assert $v_{new} < v_{old}$; assume false
})
```

The expanded encoding is as follows in IVL1:

```
$v_{old} := v$;
assert $v_{old} \geq 0$;
assert $I$;
var $\overline{x_{new}}$: $\overline{T_x}$;
$\overline{x}$ := $\overline{x_{new}}$;
assume $I$;
{
    assume $b_1$;
    encode($C_1$);
    assert $I$;
    $v_{new} := v$;
    assert $v_{new} < v_{old}$;
    assume false
} [] {
    assume $\neg b_1$;
    {
        assume $b_2$;
        encode($C_2$);
        assert $I$;
        $v_{new} := v$;
        assert $v_{new} < v_{old}$;
        assume false
    } [] {
        assume $\neg b_2$;
        ...
            {
                assume $b_n$;
                encode($C_n$);
                assert $I$;
                $v_{new} := v$;
                assert $v_{new} < v_{old}$;
                assume false
            } [] {
                assume $\neg b_n$
            }
    }
}
```

Passing example:

```
method sumn(n: Int): Int
    requires n >= 0
    ensures result == n * (n + 1)/2
{
    var acc: Int := 0;
    var i: Int := 0;
    loop
        invariant i >= 0
        invariant i <= n
        invariant acc == i * (i + 1) / 2
        decreases n - i
    {
        i < n =>
            i := i + 1;
            acc := acc + i
    };
    assert i == n;
    return acc
}
```

Failing examples: Here we have a first example that fails in the annotated line because the variant individuated in `n + i` after each iteration is not less than the precedent value.

```
method sumn(n: Int): Int
    requires n >= 0
    ensures result == n * (n + 1)/2
{
    var acc: Int := 0;
    var i: Int := 0;
    loop
        invariant i >= 0
        invariant i <= n
        invariant acc == i * (i + 1) / 2
        // @CheckError
        decreases n + i
    {
        i < n =>
            i := i + 1;
            acc := acc + i
    };
    assert i == n;
    return acc
}
```

In this other example, again the variant fails but for another reason: it is not specified a lower bound for the value of `i`, so after some iterations `assert i` $\geq$ `0` does not hold.

```
method client()
{
    var i: Int := 30;
    loop
        invariant i <= 30
        // @CheckError
        decreases i
    {
        true =>
            i := i - 1
    }
}
```

Extension Feature 9: global variables (2$\star$)

Global variables are simply variables that can be read by all methods. A method is not allowed to directly (or indirectly through method calls) modify a global variable that they do not mention in their **modifies** specification. The value that a global variable had when a method was entered can be referred to as **old**(x) and can be used in post-conditions of methods. In our understanding of the global variables it is not possible to give it a value when it is declared, this is only possible inside methods.

Special behavior It is possible to check if a method tries to modify a global variable that it has not mentioned in its **modifies** specification with translating to IVL. This is therefore done before we even try to translate the program. In future sections it will therefore be assumed that **modifies** is specified correctly.

Operational Semantics Since global variables behave like local variables the only operational semantics we have for them is their declaration:

$$\frac{}{\langle\, \text{global } x : T, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \textbf{done}, \mathfrak{m} \,\rangle}\ \text{declare-global}$$

Program Logic There are no new program logic for global variables since the keyword old is not a standalone command it only has an affect on our automation.

Automation To automate the verification we need to change in both how methods and method calls are verified. For this we just use the encoding from the slides except that our IVL1 does not have the **havoc** keyword.

Encoding for methods: When encoding methods there is now a new step which needs to be done before the body of the method is encoded. This is because we need to havoc all global variables that are modified by the method, called $\overline{m}$ below. This is done simply by assigning them to fresh variables:

```
var $\overline{m_{new}}$: $T$;
$m$ := $m_{new}$;
```

Encoding for method calls:
When a method is called the return value might be stored in a variable if this is the case we need to havoc the global variable and that variable. If this is not the case we only need to havoc the global variables.

```
$\overline{\texttt{var p:T}}$ := $\overline{e}$; // Fresh variables for input expressions
$\overline{\texttt{var o:R}}$ := $\overline{h}$; // Fresh variables for globals
assert $F[\overline{x}$ := $\overline{\texttt{p}}$];
// Havoc for global variables from modified of the method
var $\overline{h_{new}}$: $\overline{T_h}$;
$\overline{h}$ := $\overline{h_{new}}$;
// Havoc of the variable where return is stored
var $z_{new}$: $T_z$;
$z$ := $z_{new}$;
assume G[$\overline{x, \texttt{old}(h)}, z$ := $\overline{p, o}, z_{new}$]
```

Passing example:

```
global counter : Int

method inc()
    modifies counter
    ensures counter == old(counter) + 1
{
    counter := counter + 1
}

method client6()
    modifies counter
{
    counter := 0;

    assert counter == 0;
    inc(); // modifies counter without returning a value
    assert counter == 1;
    inc();
    assert counter == 2;
    inc();
    assert counter == 3
}
```

Failing example:

```
global counter : Int

method inc()
    modifies counter
    // @CheckError
    ensures counter == old(counter) + 1
{
    counter := counter + 2
}

method client6()
    modifies counter
{
    counter := 0;

    assert counter == 0;
    inc(); // modifies counter without returning a value
    assert counter == 1;
    inc();
    assert counter == 2;
    inc();
    assert counter == 3
}
```

Extension Feature 10: early return ($2\star$)

We support the use of (multiple) `return` commands in the middle of the method's body.

Operational Semantics To define the operational semantics of return statements, we define a new special symbol **returned** to be used in configuration pairs $\langle C, m \rangle$ of the program, i.e. pairs such as $\langle \mathbf{returned}, m \rangle$. We then modify existing operational semantic rules of a sequence to allow program execution with the desired behavior of return since "returned" configurations are not final configurations. We also need to modify method inlining function (i.e. **inline**$[\![z := \texttt{foo}(\overline{e})]\!]$) as follows:

```
inline⟦z := foo(ē)⟧:
    var z' : S (fresh)
    begin
        var x̄ : T̄ := ē
        C
        special
        z' := retval
    end
    z := z'
```

$$\frac{}{\langle \text{return } E, \mathfrak{m} \rangle \;\Rightarrow\; \langle \mathbf{returned}, \mathfrak{m}[\mathbf{retval} \leftarrow [\![E]\!](\mathfrak{m})] \rangle}\ \text{return}$$

$$\frac{\langle C, \mathfrak{m} \rangle \;\Rightarrow\; \langle \mathbf{returned}, \mathfrak{m}' \rangle \qquad C' \text{ does not begin with } \mathbf{special}}{\langle C; C', \mathfrak{m} \rangle \;\Rightarrow\; \langle \mathbf{returned}, \mathfrak{m}' \rangle}\ \text{seq-returned}$$

$$\frac{\langle C, \mathfrak{m} \rangle \;\Rightarrow\; \langle \mathbf{returned}, \mathfrak{m}' \rangle \qquad C' \text{ begins with } \mathbf{special}}{\langle C; C', \mathfrak{m} \rangle \;\Rightarrow\; \langle C', \mathfrak{m}' \rangle}\ \text{seq-special}$$

$$\frac{}{\langle \text{special}, \mathfrak{m} \rangle \;\Rightarrow\; \langle \mathbf{done}, \mathfrak{m} \rangle}\ \text{special}$$

Program Logic If G is the postcondition of the method, we have the following rule:

$$\frac{}{\vdash \{\{\ G[\mathbf{result} \leftarrow E]\ \}\}\ \texttt{return}_j\ E\ \{\{\ G[\mathbf{result} \leftarrow E]\ \}\}}\ \text{return}$$

Formally, to prove methods we modify the following rule:

$$\frac{\forall \texttt{return}_j \in \mathit{body}(\texttt{foo})\ \vdash \{\{\ G[\mathbf{result} \leftarrow E]\ \}\}\ \texttt{return}_j\ E\ \{\{\ G[\mathbf{result} \leftarrow E]\ \}\}}{\vdash \{\{\ F\ \}\}\ \texttt{method foo}(\overline{x} : \overline{T})\ :\ \ S\ \{\{\ G\ \}\}}\ \text{method}$$

Automation We translate a return statement to IVL1 by asserting the postconditions with **result** replaced by the returned expressions. This allows us to check the postconditions of the method hold at any return point. Then we assume false to ignore everything afterward as they are not executed.

```
assert G[result ← E]
assume false
```

Passing example:

```
method sumn(n: Int): Int
    requires n >= 0
    ensures result == n * (n + 1)/2
    decreases n
{
    match {
        n == 0 =>
            return 0,
        true =>
            var res: Int;
            res := sumn(n - 1);
            return res + n
    };
    assert false // not reachable, passes
}
```

Failing example:

```
method M(n: Int) : Int
  ensures n == 1 ==> result == 2
  ensures n != 1 ==> result == 3
{
  match {
    n == 1 =>
        // fails here and indicates it does not comply with the first ensures
        return 9
  };
  return 3
}
```

Passing example:

```
method M(n: Int) : Int
  ensures (n > 100 ==> result == n - 10) && (n <= 100 ==> result == 91)
  decreases 101 < n ? 0 : 101 - n
{
  match {
    n > 100 =>
      return n - 10,
    true =>
      var r: Int;
      r := M(n + 11);
      r := M(r);
      return r
  };
  assert false
}
```

Extension Feature 11: breaking loops (4⋆)

We have made some decisions about how we want to interpret `continue` and `break` because we only want them to work for the `loop` command and not for `for` commands. We have therefore decided that `continue` and `break` cannot appear in the body of a `for`. In the special case where a `loop` is inside of a `for` it is valid to have `continue` and `break` inside of the `loop` even though this is in principle the body of the `for`.

Operational Semantics

When we see a `break` we need to skip the rest of the body of the loop. To achieve this we need to modify the rules for `loop` and `sequence` along with introducing a new rule for break. These new semantics will make it such that when a `break` is encountered it will lead to a new state called **broke**. This state will bubble up trough `sequence` without executing following commands. Once the recursion has back-tracked to a nearest `loop` the loop will translate the **broke**state to no more iterations and go to the **done**state. The operational sematic therefore looks like this:

$$\frac{}{\langle\, \mathbf{break}, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \mathbf{broke}, \mathfrak{m} \,\rangle}\ \text{break}$$

$$\frac{\langle\, C_1, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \mathbf{broke}, \mathfrak{m}' \,\rangle}{\langle\, C_1; C_2 \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \mathbf{broke}, \mathfrak{m}' \,\rangle}\ \text{seq-break}$$

$$\frac{\forall j < i\ \ \mathfrak{m} \not\models b_j \qquad \mathfrak{m} \models b_i \qquad \langle\, C_i, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \mathbf{broke}, \mathfrak{m}' \,\rangle}{\langle\, \text{loop}\{b_1 \Rightarrow C_1, ..., b_n \Rightarrow C_n\}, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \mathbf{done}, \mathfrak{m}' \,\rangle}\ \text{loop-break}$$

For `continue` we do something similar to `break` but instead of the `loop` terminating we just do another iteration so the semantics for `continue` is:

$$\frac{}{\langle\, \mathbf{continue}, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \mathbf{continued}, \mathfrak{m} \,\rangle}\ \text{continue}$$

$$\frac{\langle\, C_1, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \mathbf{continued}, \mathfrak{m}' \,\rangle}{\langle\, C_1; C_2 \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \mathbf{continued}, \mathfrak{m}' \,\rangle}\ \text{seq-continue}$$

$$\frac{\forall j < i\ \ \mathfrak{m} \not\models b_j \qquad \mathfrak{m} \models b_i \qquad \langle\, C_i, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \mathbf{continued}, \mathfrak{m}' \,\rangle}{\langle\, \text{loop}\{b_1 \Rightarrow C_1, ..., b_n \Rightarrow C_n\}, \mathfrak{m} \,\rangle \;\Rightarrow\; \langle\, \text{loop}\{b_1 \Rightarrow C_1, ..., b_n \Rightarrow C_n\}, \mathfrak{m}' \,\rangle}\ \text{loop-continue}$$

Program Logic To prove programs containing `break` and `continue` we need logic rules for those keywords along with a change to our `loop` rule. When a `break` and `continue` are encountered, the invariant of the loop should already be satisfied because this marks the end of a loop iteration. To carry over the needed partial context after each break to after the loop, we allow proving context H_j for the j-th `break`. This gives us the following rules:

$$\frac{}{\vdash \{\{\ I \wedge H_j\ \}\}\ \texttt{break}_j\ \{\{\ I \wedge H_j\ \}\}}\ \text{break}$$

$$\frac{}{\vdash \{\{\ I\ \}\}\ \texttt{continue}\ \{\{\ I\ \}\}}\ \text{continue}$$

Because `break` can make a loop terminate without all its conditions being false we need a new rule for `loop`. This rule still needs to encapsulate that the invariants hold before and after the loop. But it must also allow for the post-condition to be something else other than the invariant and the negation of all the conditions. Namely, to allow contexts at breakpoints carry over after the loop, we have changed the post-condition resulting in the following rule:

$$F_1 := \forall \texttt{break}_j \in \mathit{body}(\bigcup_i C_i) \vdash \{\{\ I \wedge H_j\ \}\}\ \texttt{break}_j\ \{\{\ I \wedge H_j\ \}\}$$
$$F_2 := \vdash \{\{\ I\ \}\}\ \text{match}\{b_1 \Rightarrow C_1, b_2 \Rightarrow C_2, ..., b_n \Rightarrow C_n\}\ \{\{\ I\ \}\}$$
$$F_3 := \forall \texttt{continue}_j \in \mathit{body}(\bigcup_i C_i) \vdash \{\{\ I\ \}\}\ \texttt{continue}_j\ \{\{\ I\ \}\}$$

$$\frac{F_1 \qquad F_2 \qquad F_3}{\vdash \{\{\ I\ \}\}\ \text{loop}\{b_1 \Rightarrow C_1, b_2 \Rightarrow C_2, ..., b_n \Rightarrow C_n\}\ \{\{\ I \wedge \big((\bigwedge_{1\leq i\leq n} \neg b_i) \vee \bigvee_j H_j\big)\ \}\}}\ \text{loop}$$

Automation When it comes to automating the verification for `continue` and `break` there are multiple considerations:

- Encoding of `break` itself.
- Encoding of `continue` itself.
- The changes to the `loop` encoding.

As a result of our program logic rules the encoding for `break` and `continue` are rather simple.
The encoding for `break` is just that we assert the invariant of the loop and then assume false to ignore the rest of the loop body. Here there is a detail which is the keyword **broke** which can be used in invariants. This keyword should be replace by true when the loop broke and otherwise false. So in IVL1 `break` is translated to:

```
asssert I; // Where broke is replaced by true
assume false;
```

The encoding for `continue` is a bit different because it also requires us to check that the variant for the loop decreased and when this asserts the invariants **broke** should be replaced by false. The encoding in IVL1 is therefore:

```
asssert I; // Where broke is replaced by false
assert variant < variant_0;
assume false;
```

The encoding for `loop` also needs to change because now there are more than one way a loop can exit, previously it could only exit if all conditions were false. This is also why we need to have a G in the modified program logic for `loop`. To autmoate this verification we create a non-deterministic choice for each path that can be taken to a `break`.

Abstract encoding would be:

```
assert $I$;
havoc $\overline{x}$;
assume $I$;
non-deterministic-choice(
    encode(match {
        $b_1 \Rightarrow C_1$; assert $I$; assume false,
        $\cdots$,
        $b_n \Rightarrow C_n$; assert $I$; assume false
    }),
    break-path$_1$,
    break-path$_2$,
    ...
    break-path$_n$,
)
```

When expanded to IVL1 the encoding becomes:

```
assert $I$;
var $\overline{x_{new}}$: $\overline{T_x}$;
$\overline{x}$ := $\overline{x_{new}}$;
assume $I$;
{
    {
        {
            assume $b_1$;
            encode($C_1$);
            assert $I$;
            assume false
        } [] {
            assume $\neg b_1$;
            {
                assume $b_2$;
                encode($C_2$);
                assert $I$;
                assume false
            } [] {
                assume $\neg b_2$;
                ...
                    {
                        assume $b_n$;
                        encode($C_n$);
                        assert $I$;
                        assume false
                    } [] {
                        assume $\neg b_n$
                    }
            }
        }
    } [] {
        encode(break-path$_1$);
    }
} [] {
    encode(break-path$_2$);
}
...
```

With this encoding of the break paths we can verify things after the loop that would not be possible with the encoding we had before. Below is an example that can verify something about a variable based on which break path was taken which again depends on an input to the function. In other words this allows us to verify things after the loop which might depend on program state before the loop.

Example of multiple break paths In this example the loop can only break if the given input n has a remainder of 0 or 1 when modulo 2 is applied to it. In all other cases it runs forever and in those cases we cannot assert anything meaningful after the loop. However in the cases where it does break we are able to verify the state of j depending on how it broke.

```
method client(n: Int)
{
    var i: Int := 30;
    var j: Int := i;
    loop
    {
        i == 15 =>
            match {
                n % 2 == 0 =>
                    j := -1;
                    break,
                n % 2 == 1 =>
                    j := -2;
                    break
            };
            i := i - 1,
        true =>
            i := i - 1
    };

    match {
        n % 2 == 0 =>
            assert j == -1,
        n % 2 == 1 =>
            // This fails because j is actually -2
            assert j == -3,
        n % 2 == 2 =>
            // This does not fail because it is not reachable
            assert false
    };

     // This is reachable so it also fails
    assert false
}
```

Passing example for break from assignment

```
method client()
{
    var i: Int := 30;
    loop
        invariant i >= 15
        invariant i <= 30
        decreases i
    {
        true =>
            match {
                i == 15 => break
            };
            i := i - 1
    };
    assert i == 15
}
```

Failing example for break Here the break does not satisfy the invariant which says that if the loop broke then i should be 10.

```
method client()
{
    var i: Int := 30;
    loop
        invariant i >= 15
        invariant i <= 30
        invariant broke ==> i == 10
        decreases i
    {
        true =>
            match {
                i == 15 =>
                // Fails here
                break
            };
            i := i - 1
    };
    assert i == 15
}
```

Passing example for continue invariant Where continue is not marked because the invariant is satisfied even when `continue` is executed:

```
method client(n: Int): Int
    requires n >= 0
    ensures result == n * (n + 1) / 2
{
    var acc: Int := 0;
    var i: Int := -10;
    loop
        invariant i <= n
        invariant i < 0 ==> acc == 0
        invariant i >= 0 ==> acc == i * (i + 1) / 2
    {
        i < n =>
            i := i + 1;
            match {
                i < 0 =>
                    continue,
            };
            acc := acc + i

    };
    assert i == n;
    assert acc == n * (n + 1) / 2;
    return acc
}
```

Failing example for continue invariant Where continue is marked with error because the invariant does not hold after once `continue` is hit:

```
method client(n: Int)
    requires n >= 10
    requires n % 2 == 0
{
    var i: Int := 0;
    loop
        invariant i >= 0
        invariant i % 2 == 0
    {
        i < n =>
            match {
                i == 6 =>
                    i := i + 1;
                    // Fails because invariant is not satisfied
                    continue
            };
            i := i + 2
    }
}
```

Passing example for continue variant Where continue is marked not marked because variant is decreased correctly before `continue` is hit:

```
method client(n: Int)
    requires n >= 10
{
    var i: Int := n;
    loop
        invariant i >= 0
        decreases i
    {
        i > 2 =>
            match {
                i == 6 =>
                    i := i - 4;
                    continue
            };
            i := i - 2
    }
}
```

Failing example for continue variant Where continue is marked with error because the variant is not decreased once `continue` is hit:

```
method client(n: Int)
    requires n >= 10
{
    var i: Int := n;
    loop
        invariant i >= 0
        decreases i
    {
        i > 2 =>
            match {
                i == 6 =>
                    i := i + 1;
                    // Fails because variant does not descrease
                    continue
            };
            i := i - 2
    }
}
```

Technical details

We have developed our solution across Windows and Mac computers so we have seen some issues with line endings when running tests. If you run into any of these make sure you are using `LF` and `CRLF`.