A Compiler for the Verification of a C-Like Programming Language: User Manual

EECS 4302 – Compilers and Interpreters

Koko Nanah Ji
215168057
koko96@my.yorku.ca
EECS: koko96

Ege Cakmak
215173131
cakmake@my.yorku.ca
EECS: cakmake

Lassonde School of Engineering

York University

Table of Contents

1 Input Programming Language 4

1.1 Overall Structure of a Program 4

1.1.1 Boolean and Integer Primitive Types 4

1.1.2 Variable Declaration and Assignment 5

1.1.3 Basic Type Checking 5

1.1.4 Conditional Statements (if/else) with Contracts (require/ensure) 5

1.1.5 Functions and Functions Calls (with Contracts) 11

1.1.6 Non-nested Loops (with Contracts) 7

1.1.7 Lists (w/o Indexing) and Quantification Operations over Lists (all/some) 7

1.2 How a Specification is Written 8

1.3 List of Advanced Programming Features 9

1.3.1 Feature 1: Functions with Local Scopes and Function Calls (with Contracts) 9

1.3.2 Feature 2: Non-Nested Loops (with Contracts) 11

1.3.3 Feature 3: Lists (w/o Indexing) and Quantification Operations over Lists (all/some) 11

2 Output Specification Language 16

2.1 General Structure of Output 17

2.2 Output Feature Translations 17

2.2.1 Conditional Statements (if/else) with Contracts (require/ensure) 17

3 Examples 19

3.1 Example Input – If Statement 20

3.2 Example Input – Double Nested If Statement 22

3.3 Example Input – Triple Nested If Statement 24

3.4 Example Input – Functions 26

3.5 Example Input – Function with an If Statement 28

3.6 Example Input – Loop 30

3.7 Example Input – Loop 32

3.8 Example Input – Lists and Quantification Operations 34

3.9 Example Input – Function Calling Loop 36

3.10 Example Input – Chef's Special 38

3.11 Example Input – Incorrect 3.1 40

3.12 Example Input – Incorrect 3.7 42

3.13 Example Input – Incorrect 3.10 44

4 Miscellaneous Features 48

4.1 Type Checking 48

4.2 Error Reporting 48

4.3 Allowing Explicit Boolean Values in Alloy 48

5 Limitations 49

1 Input Programming Language

1.1 Overall Structure of a Program

The input programming language that will be translated into Alloy supports the below features.

- Boolean and Integer Primitive Types (with Operations)
- Variable Declaration and Assignment
- Basic Type Checking
- Conditional Statements (if/else) with Contracts (require/ensure)
- Functions with Local Scopes and Function Calls (with Contracts)
- Non-nested Loops with Contracts
- Lists (w/o Indexing) and Quantification Operations over Lists (all/some)

The syntax of the input programming language is like modern programming languages such as (C, Python etc.) except for couple of exceptions. The output of the program is stored in output.ke file in the high level directory.
The rest of this subsection describes and briefly explains each bullet point above.

1.1.1 Boolean and Integer Primitive Types

The input programming language supports only Integer and Boolean primitive data types. Keywords “int” and “bool” are reserved to enable users to declare their own variables. Integer operations such as add, subtract, divide, multiply, modulo are supported as well as Boolean operations such as and, or, implies, equality, inequality and negation.

```
int var1;
int var2 = 1;
int var3 = 0;
var1 = var1 + var 2;   // Addition Example

bool var5;                    // Uninitialized Boolean Variable Declaration
bool var6 = true;             // Initialized Boolean Variable Declaration
bool var 7 = false;
var 5 = var6 && true;  // Conjunction Example
```

Figure 1: Demonstration: Primitive Types

1.1.2 Variable Declaration and Assignment

The input programming language supports declaration of variables and initializing their values using either a constant or value of another variable. It is also worth noting here that sequential assignments are supported as will be shown in the comments.

```
int var1;                 // Uninitialized Integer Variable Declaration
int var2 = 1;             // Initialized Integer Variable Declaration
int var3 = 0;
int var4 = -1;
var1 = var3;              // Copying Variables

bool var5;                // Uninitialized Boolean Variable Declaration
```

```
bool var6 = true;        // Initialized Boolean Variable Declaration
bool var 7 = false;
var5 = var6;                // Copying Values of Variables
```

Figure 2: Demonstration: Variable Declaration and Assignment

1.1.3 Basic Type Checking

The input programming language also supports type checking of variables during translation. Uninitialized variables will also be detected for type checking therefore our compiler can warn the user in case they attempt to access uninitialized variables.

```
int var1;
int var2 = 1;
bool var5;
bool var6 = true;
var1 = var5;              // WILL FAIL
var2 = var1;              // WILL FAIL
var6 = var5;              // WILL FAIL
```

Figure 3: Demonstration: Type Checking

1.1.4 Conditional Statements (if/else) with Contracts (require/ensure)

The input programming language also supports if/else statements with or without contracts. Nested if statements are allowed as well. (will be demonstrated in later sections)

```
int var1 = 1;

if (var1 == 1) {
    var1 = 5;
} else {
    var1 = 10;
}

bool var2 = false;
if_require (var1 == 10)         // Ensures that var1 has value 10 right before the if statement
if (var2 <=> false) {
    var1 = 20;
}
if_ensure (var1 == 20)           // Ensures that var1 has value 20 right after the if statement
```

Figure 4: Demonstration: Conditional Statements

1.1.5 Functions and Function Calls (with Contracts)

Functions can be declared and called in the input programming language. Variables in the input language are passed by values to the functions. And, every function must have a return type but can have any number of parameters.

Currently, it is enforced by design that every function has a require and ensure section. Yet, these contracts are not mandatory to implement as they can be skipped by inserting just “true” in them.

A limitation that is worthwhile to mention here is that variable declaration inside functions are not supported at the moment. Thus, these variables need to be declared at the parameters section of the function, and also, while invoking the function a placeholder argument should be passed for those arguments.

Lastly, values of variables before the translation of the function can be retrieved by appending “_old” to the names of the variables.

```
fun int fun1 (int var1) {            // Function Declaration with 1 Parameter
    fun_require(var1 > 0)           // Ensures that var1 has a positive value right before the function call.
        var1 = var1;
        return var1;
    fun_ensure(var1 == var1_old)      // Ensures that var1 is unchanged right after the function call.
}

int x = 10;
x = fun1(x);                          // Calls function fun1.

fun int fun2 (int var1, int tmp) {   // Function Declaration with 1 Actual and 1 Extra Parameter
    fun_require(var1 > 0)           // Ensures that var1 has a positive value right before the function call.
        tmp = var1 + 1;
        return tmp;
    fun_ensure(var1 == var1_old)      // Ensures that var1 is unchanged right after the function call.
}

int y = 10;
int z;
x = fun2(y,z);                        // Calls function fun2. Note that this will fail since z is uninitialized yet.
```

Figure 5: Demonstration: Functions

1.1.6 Non-nested Loops (with Contracts)

The input programming language supports loops for statements that repeat. The syntax of loops in the input programming language is a bit different than the ordinary programming languages. Contracts are supported in loops and users are required to implement the invariant, variant, and initialization sections separately along with the implementation body.

```
int sum;
int current;
loop_require(true)          // Precondition need to be implemented.
loop_init{                  // Initialization need to be implemented.
       sum = 0;
    current = 3;
}
loop(current != 0) {  // Exit condition goes here
loop_invariant((sum == (3-current)) && (current >= 0) && (current < 4))
loop_variant(current) // Loop variant and invariant need to be implemented as well.
sum = sum + 1;        // Implementation body starts here.
current = current - 1;
}
loop_ensure(sum == 3)  Postcondition need to be implemented.
```

Figure 6: Demonstration: Loops

1.1.7 Lists (w/o Indexing) and Quantification Operations over Lists (all/some)

The input programming language also supports lists of primitive types.

Currently, each list must have a primitive type and type of each element in lists must comply with its list's primitive type. The current state of lists only supports "add" and "remove" operations. However, quantification operations over lists are supported and these operators can be used as demonstrated below. The "each" keyword is reserved for quantification operations and can only be used in inside "all" / "some" like below.

```
int[] list1;                        // Creates an array of integers.
list1.add(1);                       // Adds integers to the list.
list1.add(2);
list1.add(3);
list1.add(true);                    // WILL FAIL.

if_require(list1.all(each > 0))     // Ensures list only has positive numbers right before the if statement.
if (list1.some(each == 1)) {        // Adds integer 4 to the list if integer 1 exists in the list.
    list1.add(4);
}
if_ensure(list1.some(each == 1))    // Ensures list has integer 1 right after the if statement.
```

Figure 7: Demonstration: Lists and Quantification Operators

1.2 How a Specification is Written

The input programming language supports verification of all features mentioned in the previous section. Due to the declarative nature of Alloy though, our compiler only translates the statements with contracts as there is nothing to verify otherwise. For instance, there is nothing to verify when the only operation is a variable declaration.

Below is the list of arithmetic operators in the input language sorted by higher to lower precedence.

- *
- /
- %
- +
- -

Below is the list of logical operators in the input language sorted by higher to lower precedence.

- !
- &&
- ||
- =>
- <=> (Logical equality)

Below keywords are reserved for the input programming language and cannot be used as a variable name.

- if, else, if_require, if_ensure
- loop, loop_require, loop_ensure, loop_init, loop_invariant, loop_variant
- fun, fun_require, fun_ensure, return
- true, false, int, bool, int[], bool[]
- each, all, some
- int, bool, true, false
- _old (except for variable names in post conditions)

Below keywords are specifically used for specification. These keywords can only be used in their respective places as suggested by their names. (eg. if_require can only be used for enforcing preconditions for if statements etc.)

- if_require, if_ensure (Inside if statements)
- loop_require, loop_ensure (Inside loops)
- invariant, variant (Inside loops)
- fun_require, fun_ensure (Inside functions)

All the above keywords for specification can be used to verify the correctness of the input programming language.

The specification of each specification construct can be either propositional or in predicate logic depending on user's wish as our compiler makes a singleton data structure for all available variables within the scope and their respective values.

As demonstrated in section 1.1.5, the keyword "old" can be used to retrieve the pre-state of variables, if the implementation block introduces any changes to the value of the variables.

Since the only way variables' values can change is through assignments, our compiler keeps record of changes every time it encounters an assignment. The compiler will also create new variables for the new values of the variables so that the original value of the variable can be preserved. Although this process is not explicitly visible to the user during the translation, these new variables will be available to the user in the output code. These new variables will store the post-states' of variables, and their name will be the name of the pre-state followed by a single quote (‘). There are no restrictions on the number of post-states. (i.e. arg1''' is possible)

e.g. arg1 is the original name, arg1' is the post-state

It is currently possible to write fairly complex specification constructs thanks to all the operators and features of the input programming language. We will not be giving any specific complex examples here as there are already many examples in the following sections and chapters.

1.3 List of Advanced Programming Features

1.3.1 Feature 1: Functions with Local Scopes and Function Calls (with Contracts)

The syntax of functions is much similar to modern programming languages as demonstrated in 1.1.5. We will not be reviewing the syntax as it has already been presented to reader's pleasure earlier.

The current state of the compiler seems to be able to generate sound outputs. The Alloy Analyzer seems to find counterexamples when a logically correct program is tampered to be incorrect on purpose. Here are two code snippets with translations where one is correct and the other is incorrect.

```
fun int add5(int x){
      fun_require(true)
                  x = x + 5;  // Adds 5 to x as the function name suggests
                  return x;
      fun_ensure(x == (x_old + 5)) // Contract is valid
}

int x = 10;
int xplus5 = add5(x);
bool result = (x + 5) == xplus5;
```

Figure 8: Function Example Input Programming Language (Logically Correct)

```
open logicFuncs
```

```
pred predFunction0 [arg1,arg1':Int] {
       ((True) in True) and arg1'= arg1.add[5]  // corresponds to the assignment at line 3
       ((((arg1' = arg1.add[5]) => True else False)) in True)  // post condition
}

fun funFunction0 [arg1,arg1':Int] : Int {
       { return : Int | ((True) in True) and arg1' = arg1.add[5] and return = arg1' }
}

check assertFunction0 {
        { all arg1:Int | some arg1':Int | ((True) in True)  => predFunction0[arg1,arg1'] }
}
```

Figure 9: Function Example Output Specification Language (Logically Correct)

```
fun int add5(int x){
       fun_require(true)
                     x = x - 5;  // Mistakenly subtracts 5 instead of adding
                     return x;
       fun_ensure(x == (x_old + 5))  // Contract is invalid
}

int x = 10;
int xplus5 = add5(x);
bool result = (x + 5) == xplus5;
```

Figure 10: Function Example Input Programming Language (Logically Incorrect)

```
open logicFuncs

pred predFunction0 [arg1,arg1':Int] {
       ((True) in True) and arg1'= arg1.sub[5]
       ((((arg1' = arg1.add[5]) => True else False)) in True) // post condition
}

fun funFunction0 [arg1,arg1':Int] : Int {
       { return : Int | ((True) in True) and arg1'=arg1.sub[5] and return = arg1' }
}

check assertFunction0 {
        { all arg1:Int | some arg1':Int | ((True) in True)  => predFunction0[arg1,arg1'] }
}
```

Figure 11: Function Example Output Specification Language (Logically Incorrect)

We would like to briefly explain here how the input programming language is translated into the output specification language at a high-level, before we debate whether our transformation is semantics-preserving and correct.

We are going to be explaining examples in Figure 10 and 12 at once since the only difference in between is the operator at line 3.

Both snippets in Figure 10 and 12 aim to do the same thing, adding 5 to a given integer and returning it. However, the code snippet in Figure 12 mistakenly subtracts 5 instead of adding.

We will be examining the body of the check statement first and then examine the bodies of the predicate and the function.

In the output specification language (Figure 11 and 13), we can see 2 variables getting declared in the check body. “arg1” refers to the pre-state of variable “x” whereas “arg1’” refers to the post-state. This check body basically states that for all possible pre-states of “x”, there exists a post-state of “x” where the function predicate holds if the precondition is true. The left-hand side of the implication statement is our precondition (it was “true” in the input programming language) which is translated to ((True) in True) which basically means true in Alloy. In this case the statement in fact makes sense, it is saying that if the precondition holds then the predicate for the function should hold as well!

The predicate ***pred predFunction0*** is the translation of the function with contracts. It states that if the assignments are logically correct then the post condition must hold. We can see the assignment (x = x +5) being done in Alloy (line 4 of Figure 11 and 13) and we can also see the post condition (line 5 of Figure 11 and 13). Note that this predicate looks structurally intact. When we try verifying both outputs (both correct and incorrect) we can clearly observe the correct one passing and the incorrect one failing with a counterexample found by the Alloy Analyzer.

The function ***fun funFunction0*** is generated to be used by Alloy for subsequent function calls to this function from other Alloy predicates.

We believe that our transformation is semantics-preserving since we can clearly explain the transformation in a way that makes sense. Also, the fact that Alloy Analyzer is able to find a counterexample when we deliberately break our model allows us to convince ourselves that the translation is valid thus it is semantics-preserving.

Finally, we should note here that the input language does not support variable declaration inside functions. It is a requirement that all required variable declarations are done before the function declaration.

1.3.2 Feature 2: Non-Nested Loops with Contracts

The syntax of loops in our input language is a bit different than standard loop syntax, as can be observed at Figure 5. We will not be explicitly explaining the syntax of loops here as it has been already explained in section 1.1.6, but we can note here to remind the reader that our input

language syntax allows user to implement the variant, invariant, initial step and body of the loop separately.

Currently we believe that our compiler can translate a given loop in our input language to valid and accurate Alloy Analyzer code. Conversely, when we tamper to make the input logically incorrect, our code still compiles but Alloy Analyzer is unable to find any counter examples suggesting issues in the output specification code. However, we will not be fixing it since fixing/learning semantics of Alloy is beyond the scope of this project.

We will be attempting to strengthen this hypothesis in the rest of this subsection. Here are two not so simple looking code snippets that aim to add all numbers from 4 to 1.

```
int sum;
int current;
loop_require(true)
loop_init{
      sum = 0;
      current = 4;
}
loop(current < 5) {
loop_invariant((current >= 0) && (current < 5) )
loop_variant(current)
sum = sum + current;
current = current - 1;
}
loop_ensure(sum == 10)
```

Figure 12: Loop Example Input Programming Language (Logically Correct)

```
open logicFuncs

pred predForStatement0 [arg2, arg2’, arg2'', arg2''':Int, arg1, arg1', arg1'':Int] {
      (((True) in True) and arg1'= 0 and arg2'' = 4 and arg2' = 4
      =>
      (andGate[((arg2'' >= 0) => True else False), ((arg2'' < 5) => True else False)] in
True))

      ((andGate[((arg2 >= 0) => True else False), ((arg2 < 5) => True else False)] in True)
       and(((arg2 < 5) => True else False) in True ))
      =>
      ((True) in True) and arg1'' = arg1.add[arg2] and arg2''' = arg2.sub[1]
      =>
      (andGate[((arg2''' >= 0) => True else False), ((arg2''' < 5) => True else False)] in
True and  (arg2''' >= 0) and (arg2 > arg2'''))

      ((andGate[((arg2 >= 0) => True else False), ((arg2 < 5) => True else False)] in True)
and not((((arg2 < 5) => True else False) in True))
       =>
      ((((arg1 = 10) => True else False)) in True))     // post condition
```

```
}

check assertForStatement0 {
       { all arg2:Int,arg1:Int | some arg2':Int,arg2'':Int,arg2''':Int,arg1':Int,arg1'':Int
| ((True) in True)  => predForStatement0[arg2,arg2',arg2'',arg2''',arg1,arg1',arg1''] }
}
```

Figure 13: Loop Example Output Specification Language (Logically Correct)

```
int sum;
int current;
loop_require(true)
loop_init{
    sum = 0;
    current = 4;
}
loop(current < 5) {
loop_invariant((current >= 0) && (current < 5))
loop_variant(current)
sum = sum - current;
current = current - 1;
}
loop_ensure(sum == 10)
```

Figure 14: Loop Example Input Programming Language (Logically Incorrect)

```
open logicFuncs

pred predForStatement0 [arg2,arg2',arg2'':Int,arg1,arg1',arg1'':Int] {
      (((True) in True) and arg1'= 0 and arg2'= 4
      =>
      (andGate[((arg2' >= 0) => True else False), ((arg2' < 5) => True else False)] in
True))

      ((andGate[((arg2 >= 0) => True else False), ((arg2 < 5) => True else False)] in True)
       and(((arg2 < 5) => True else False) in True))
      =>
      ((True) in True) and arg2'' = arg2.sub[1] and arg1'' = arg1.sub[arg2]
      =>
      (andGate[((arg2'' >= 0) => True else False), ((arg2'' < 5) => True else False)] in
True and (arg2'' >= 0) and (arg2 > arg2''))

      ((andGate[((arg2 >= 0) => True else False), ((arg2 < 5) => True else False)] in True)
and not((((arg2 < 5) => True else False) in True))
       =>
      ((((arg1 = 10) => True else False)) in True))  // post condition
}

check assertForStatement0 {
```

```
       { all arg2:Int,arg1:Int | some arg2':Int,arg2'':Int,arg1':Int,arg1'':Int | ((True) in
True)  => predForStatement0[arg2,arg2',arg2'',arg1,arg1',arg1''] }
}
```

Figure 15: Loop Example Output Specification Language (Logically Incorrect)

We note here that the only difference between the two snippets is the operator at line 10.

Luckily, with both samples (Figure 14 and 16) our compiler can generate code that can be compiled by the Alloy Analyzer. Unfortunately, as mentioned earlier that Alloy Analyzer fails to find a counterexample for the logically incorrect code (Figure 16). This might mean there is a semantic issue in the generated output, however we believe that the generated code is syntactically and semantically correct based on the Alloy specification. This might even be a bug in Alloy, considering the number of inconsistencies we have come across in Alloy, we will talk about some of these in the "Limitations" chapter.

In the check body of the generated code we can see two variables and their post states getting declared for the loop predicate with no precondition. At high-level, this loop predicate checks that each of the below are TRUE. (This information was taken from Jackie's 3311 Slides.)

- Establishment of Loop Invariant
- Maintenance of Loop Invariant
- Establishment of Postcondition upon Termination
- Loop Variant Stays Non-Negative Before Exit
- Loop Variant Keeps Decrementing Before Exit

We leave examining the details of the output code to the reader. Upon our examinations we believe that the predicate should be valid thus the output should be valid as well. Ultimately, the conversion should be semantics-preserving, ignoring the issue that we believe that exists in Alloy as we just talked about.

1.3.3 Feature 3: Lists (w/o Indexing) and Quantification Operations over Lists (all/some)

The primitive syntax of lists in our input language is straight forward as has been demonstrated already in section 1.1.7.

Currently, we believe that our compiler supports declaration of lists and addition / removal operations over them as well as quantification operations such as "all" / "some" for batch checking the correctness of a Boolean expression for each item over a list.

Since it is extremely easy to cause a lot of possible states with lists (state explosion), it is on the very contrary, extremely difficult to verify programs that cause state explosions.

As a result, we do not support adding to lists inside any block with contracts. Although we can still generate code that compiles, we just do not know whether it is correct or not simply because we cannot verify.
We are also aware of an issue that using existential operator (some) might cause a counterexample to be found. For instance, output generated from below code causes Alloy to

find a counter example, which does not make sense since we can easily see from Figure 18 that the postcondition is correct. This might suggest that there is a semantic error in the way we generate output. However, we will not be fixing it due to timing constraints. Yet, this could also be an issue with Alloy Analyzer itself!

```
int[] x;
x.add(1);
x.add(2);
x.add(3);
int y;

if_require(x.all(each > 0))
if(true) {
	y = 1;
}
if_ensure(x.some(each > 0))
```

Figure 16: Lists Example of an Issue Causing a Counterexample

Because of the state explosion issue and the issue regarding the existential operator we believe that we cannot do any extensive tests for this feature now. Therefore, we will not be doing any tests with correct and incorrect code snippets for this feature.

Here's a very simple working code snippet that we will use to summarize the translation of quantification operations at a high level.

```
int[] x;
x.add(1);
x.add(2);
x.add(3);
int y = 15;
if_require(x.all(each > 0))
if(true) {
	y = 4;
}
if_ensure(x.all(each > 0))
```

Figure 17: Working Very Simple Input Programming Language

```
open logicFuncs

pred predIfStatement0 [arg1:seq Int,arg2,arg2':Int] {
```

```
      ((True) in True) =>
             ((True) in True) and arg2'=4
      ((((all arrayElems: arg1.elems | arrayElems  in {each: Int | (((each > 0) => True else
False) in True)})=> True else False)) in True)                    // post condition
}

check assertIfStatement0 {
       { all arg1:seq Int,arg2:Int |  some arg2':Int | ((((all arrayElems: arg1.elems |
arrayElems  in {each: Int | (((each > 0) => True else False) in True)})=> True else False))
in True)  => predIfStatement0[arg1,arg2,arg2'] }
}
```

Figure 18: Working Very Simple Output Specification Language

```
x.all(each > 0)
```

Figure 19: Quantification Operation Input Programming Language

```
all arrayElems: arg1.elems | arrayElems  in {each: Int | (((each > 0) => True else False) in
True)})=> True else False)) in True)
```

Figure 20: Quantification Operation Specification Language

As you can see from Figure 17 and 18, translation is rather simple. Our code creates a satisfying set by looking at the statement inside the quantification operation and then in the translation it checks whether each item in the list is in the satisfying set or not.

The keyword “each” in the input language is can also be matched as either an integer or a boolean since it is already included in the grammar. (in logicalOp and arithmeticOp) Thanks to this, we can easily detect this keyword and treat it specially across the whole list not just as a single integer or a boolean.

Despite the issue with the existential operator and the state explosion, we believe that our reasoning makes sense and translation of quantification operators should be semantics-preserving as demonstrated in Figure 19 and 20.

2 Output Specification Language

In this chapter we will first briefly talk about the general structure of the output and then we will demonstrate how some of the features from 1.1 are translated.

2.1 General Structure of Output

Let's refer to the very simple example we have in Figure 20.

In every translation the first line is always "open logicFuncs". This is a statement that tells Alloy Analyzer to load our logical operation utilities library. One might wonder why we had to implement Booleans ourselves. The answer is that Alloy by default does not allow Boolean constants in the code. By implementing Booleans ourselves we are able to bypass this issue. This file (logicFuncs.als) can be found in our submission and should be placed in the same path as the Alloy code that is about to be verified.

In every translation, there is a check block and a predicate block. In the predicate block, we usually verify functions, loops, if statements or quantification operations followed by their postconditions. Variable value assignments take place in the predicate block as well.

In the check block we declare each variable in the input and the post states for the variables if necessary (i.e. if body of the statement has assignments). Also using these variables, we specify the preconditions and refer to the predicate we just mentioned to verify the model.

2.2 Output Feature Translations

As mentioned previously, due to the declarative nature of Alloy, our compiler only translates the statements with contracts as there is nothing to verify in trivial statements such as declaring a variable. In this section we will be demonstrating the output our compiler is able to generate.

Since we have already demonstrated sample output for Functions, Loops and Lists we will not be demonstrating those ones again and kindly ask the reader to refer to section 1.3. Therefore, we will now be demonstrating output for if statements, which is the only thing left we must demonstrate.

2.2.1 Conditional Statements (if/else) with Contracts (require/ensure)

Figure 23 is the output generated by our compiler when the code snippet in Figure 3 is inputted.

```
open logicFuncs

pred predIfStatement0 [arg1:Bool,arg2,arg2':Int] {
       ((((arg1 in False) => True else False)) in True) =>
              ((True) in True) and arg2'=20
       ((((arg2' = 20) => True else False)) in True)  // post condition
}

check assertIfStatement0 {
        { all arg1:Bool,arg2:Int |  some arg2':Int | ((((arg2 = 10) => True else
False)) in True)  => predIfStatement0[arg1,arg2,arg2'] }
}
```

Figure 21: Demonstration: If Statements Output

We can clearly see a predicate and a check body in the output just like we discussed in 2.1.

In the check body we have the variables and their required post states getting declared as well as the precondition and the reference to the predicate generated for the if statement.

In the predicate body, our compiler generates a logically equivalent Alloy code along with the postcondition(s).

Please notice that our compiler did not generate any output for the if statement without the contract since there is nothing to verify!

3 Examples

In this chapter we will be sharing code examples involving real-world scenarios rather than specifically demonstrating the features of the compiler. Please refer to Chapter 5 before examining these examples.

3.1 Example Input – If Statement

```
// Bank Transaction Application

int cust1Balance = 100;
int cust2Balance = 600;
int cust3Balance = 300;

// customer 3 transfers funds to customer 1
cust3Balance = cust3Balance - 200;
cust1Balance = cust1Balance + 200;

// cust3Balance deposits 300
cust3Balance = cust3Balance + 300;

// cust1Balance withdraws 200
cust1Balance = cust1Balance - 200;

// cust3 asks cust1 for a loan of 200
// this time since customer 1 does not know how much money he has, he says he will loan the money if
he has enough.
int borrowAmount = 200;
if_require(cust1Balance > borrowAmount)
if (cust1Balance > borrowAmount) {
 cust1Balance = cust1Balance - borrowAmount;
 cust3Balance = cust3Balance + borrowAmount;
}
if_ensure((cust1Balance == cust1Balance_old - borrowAmount) && (cust3Balance == cust3Balance_old +
borrowAmount))
```

Figure 22: Example Input – If Statement

```
open logicFuncs

pred predIfStatement0 [arg2:Int,arg1,arg1':Int,arg3,arg3':Int] {
      ((((arg3 > arg2) => True else False)) in True) =>
              ((True) in True) and arg1'=arg1.add[arg2] and arg3'=arg3.sub[arg2]
      ((andGate[((arg3' = arg3.sub[arg2]) => True else False), ((arg1' = arg1.add[arg2]) => True
else False)]) in True)                  // post condition
}

check assertIfStatement0 {
       { all arg2:Int,arg1:Int,arg3:Int | some arg1':Int,arg3':Int | ((((arg3 > arg2) => True else
False)) in True)  => predIfStatement0[arg2,arg1,arg1',arg3,arg3'] }
}
```

Figure 23: Example Output – If Statement

- Variable Declarations and Assignments
- If Statements with Contracts
- Arithmetic Operations
- Relational Operations
- Logical Operations
- Old syntax
    - Notice in the postcondition of the if statement that we are accessing the pre-state of some of the variables.

- This program is logically correct.
- The output program compiles and verifies with no counterexamples in Alloy.

3.2 Example Input - Double Nested If Statement

```
// Yet Another Bank Transaction Application

int cust1Balance = 100;
int cust2Balance = 600;
int cust3Balance = 300;

// customer 3 transfers funds to customer 1
cust3Balance = cust3Balance - 200;
cust1Balance = cust1Balance + 200;

// cust3Balance deposits 300
cust3Balance = cust3Balance + 300;

// cust1Balance withdraws 200
cust1Balance = cust1Balance - 200;

cust2Balance = cust2Balance + 9999;
cust1Balance = cust1Balance + 9999;

// cust3 asks cust1 for a loan of 200
// this time since customer 1 knows that he received a hefty amount of money recently, he does not
care about
// the amount. though, he says that he will only pay half and on the condition that cust2 will pay
the other half as well.
// cust3 also makes contingency plans to get a loan from the bank in case he cannot borrow any money
from the other customers
int borrowAmount = 200;
int borrowAmountHalf = borrowAmount / 2;
if_require(cust1Balance > borrowAmountHalf && cust2Balance > borrowAmountHalf)
if (cust1Balance > borrowAmountHalf) {
    if (cust2Balance > borrowAmountHalf) {
        cust1Balance = cust1Balance - borrowAmountHalf;
        cust2Balance = cust2Balance - borrowAmountHalf;
        cust3Balance = cust3Balance + borrowAmount;
    } else {
        // cust3 gets a loan from the bank
        cust3Balance = cust3Balance + borrowAmount;
    }
} else {
    // cust3 gets a loan from the bank
    cust3Balance = cust3Balance + borrowAmount;
}
if_ensure(cust3Balance == cust3Balance_old + borrowAmount)
```

Figure 24: Example Input – Double Nested If Statemen

```
open logicFuncs

pred predIfStatement0
[arg4:Int,arg1,arg1':Int,arg2,arg2',arg2'',arg2''':Int,arg3:Int,arg5,arg5':Int] {
      ((((arg5 > arg3) => True else False)) in True) =>
            ((True) in True) and (((((arg1 > arg3) => True else False)) in True) =>
            ((True) in True) and arg1'=arg1.sub[arg3] and arg5'=arg5.sub[arg3] and
arg2'=arg2.add[arg4] and arg2''=arg2'
      else
            ((True) in True) and arg2''=arg2.add[arg4] and arg1'=arg1 and arg5'=arg5)  and
arg2'''=arg2''
      else
            ((True) in True) and arg2'''=arg2.add[arg4] and arg1' = arg1 and arg5'=arg5
      ((((arg2''' = arg2.add[arg4]) => True else False)) in True)                    // post
condition
}

check assertIfStatement0 {
       { all arg4:Int,arg1:Int,arg2:Int,arg3:Int,arg5:Int | some
arg1':Int,arg2':Int,arg2'':Int,arg2''':Int,arg5':Int | ((andGate[((arg5 > arg3) => True else
False), ((arg1 > arg3) => True else False)]) in True)  =>
predIfStatement0[arg4,arg1,arg1',arg2,arg2',arg2'',arg2''',arg3,arg5,arg5'] }
}
```

Figure 25: Example Output – Double Nested If Statement

- Variable Declarations and Assignments
- If Statements with Contracts
    - Notice that we support double nested if statements as well.
- Arithmetic Operations
- Relational Operations
- Logical Operations
- Old syntax

- This program is logically correct.
- The output program compiles and verifies with no counterexamples in Alloy.

3.3 Example Input - Triple Nested If Statement

```
// Bank Transaction Application
// Takes forever to verify.
int cust1Balance = 100;
int cust2Balance = 600;
int cust3Balance = 300;
// customer 3 transfers funds to customer 1
cust3Balance = cust3Balance - 200;
cust1Balance = cust1Balance + 200;
// cust3Balance deposits 300
cust3Balance = cust3Balance + 300;
// cust1Balance withdraws 200
cust1Balance = cust1Balance - 200;
cust2Balance = cust2Balance + 9999;
cust1Balance = cust1Balance + 9999;
int bank = 99999;

// cust3 asks cust1 for a loan of 200
// this time since cust1 knows that he received a hefty amount of money recently, he does not care
about
// the amount. though, he says that he will only pay half and on the condition that cust2 will pay
the other half as well.
// cust3 also makes contingency plans to get a loan from the bank in case he cannot borrow any money
from the other customers.
// bank also decides to make a gesture and says that they will give cust3 the full amount for free
on the condition that
// cust1 and cust2 pay their halves.

int borrowAmount = 200;
int borrowAmountHalf = borrowAmount / 2;
if_require(cust1Balance > borrowAmountHalf && cust2Balance > borrowAmountHalf)
if (cust1Balance > borrowAmountHalf) {
    if (cust2Balance > borrowAmountHalf) {
        cust1Balance = cust1Balance - borrowAmountHalf;
        cust2Balance = cust2Balance - borrowAmountHalf;
        cust3Balance = cust3Balance + borrowAmount;
        if (bank > borrowAmount){
            bank = bank - borrowAmount;
            cust3Balance = cust3Balance + borrowAmount;
        }
    } else {
        // cust3 gets a loan from the bank
        cust3Balance = cust3Balance + borrowAmount;
    }
} else {
    // cust3 gets a loan from the bank
    cust3Balance = cust3Balance + borrowAmount;
}
if_ensure((cust3Balance >= (cust3Balance_old + borrowAmount)))
```

Figure 26: Example Input – Triple Nested If Statement

```
open logicFuncs

pred predIfStatement0
[arg1:Int,arg2,arg2':Int,arg3,arg3':Int,arg4,arg4',arg4'',arg4''',arg4'''':Int,arg5:Int,arg6,arg6':Int]
{
      ((((arg6 > arg5) => True else False)) in True) =>
              ((True) in True) and  ( ((((arg3 > arg5) => True else False)) in True) =>
              ((True) in True) and  ( ((((arg2 > arg1) => True else False)) in True) =>
              ((True) in True) and arg4''=arg4'.add[arg1] and arg2'=arg2.sub[arg1] )  and
arg4'=arg4.add[arg1] and arg3'=arg3.sub[arg5] and arg6'=arg6.sub[arg5] and arg4'''=arg4''
      else
              ((True) in True) and arg4'''=arg4.add[arg1] and arg3'=arg3 and arg2'=arg2 and
arg6'=arg6 )  and arg4''''=arg4'''
      else
              ((True) in True) and arg4''''=arg4.add[arg1] and arg3'=arg3 and arg2'=arg2 and
arg6'=arg6
      ((((arg4'''' >= arg4.add[arg1]) => True else False)) in True)                  // post
condition
}

check assertIfStatement0 {
       { all arg1:Int,arg2:Int,arg3:Int,arg4:Int,arg5:Int,arg6:Int | some
arg2':Int,arg3':Int,arg4':Int,arg4'':Int,arg4''':Int,arg4'''':Int,arg6':Int | ((andGate[((arg6 > arg5)
=> True else False), ((arg3 > arg5) => True else False)]) in True)  =>
predIfStatement0[arg1,arg2,arg2',arg3,arg3',arg4,arg4',arg4'',arg4''',arg4'''',arg5,arg6,arg6'] }
}
```

Figure 27: Example Output – Triple Nested If Statement

- Variable Declarations and Assignments
- If Statements with Contracts
    - Notice that we support triple nested if statements as well.
- Arithmetic Operations
- Relational Operations
- Logical Operations
- Old syntax

- This program should be correct.
- The output program compiles, but the verification takes too long to finish. Therefore, we cannot verify it.

3.4 Example Input – Functions

```
// Bank Transaction Application
// One day, miracally, the bankers at the bank finally realize that
// they can use functions instead of doing everything manually.

// However government started taxing functional transaction for deposits. that's why now each
deposit operation cause 1 dollar fee
// Customer Accounts
int cust1Balance = 100;
int cust2Balance = 600;
int cust3Balance = 300;

fun int withdraw (int amount, int balance, int result) {
    fun_require((balance >= amount) && (amount >= 0))
        result = balance - amount;
        return result;
    fun_ensure(result == balance_old-amount_old && (result>=0))
}

fun int deposit (int amount, int balance, int result) {
    fun_require(amount > 1)
    balance = withdraw (1, balance, result); // Tax
    result = balance + amount;                          // new balance
    return result;
    fun_ensure(result == balance_old + amount_old-1)
}

// customer 3 transfers funds to customer 1
cust3Balance = withdraw(200, cust3Balance, 0);
cust1Balance = deposit(200, cust1Balance, 0);

// cust3Balance deposits 300
cust3Balance = deposit(300, cust3Balance, 0);

// cust1Balance withdraws 200
cust1Balance = withdraw(200, cust1Balance, 0);
```

Figure 28: Example Input – Functions

```
open logicFuncs

pred predFunction0 [arg2,arg2':Int,arg3:Int,arg1:Int] {
       ((True) in True) and arg2'=arg1.sub[arg3]
       ((andGate[((arg2' = arg1.sub[arg3]) => True else False), ((arg2' >= 0) => True else False)]) in
True)                // post condition
}

fun funFunction0 [arg2,arg2':Int,arg3:Int,arg1:Int] : Int {
       { return : Int | ((True) in True) and arg2'=arg1.sub[arg3] and return = arg2'}
}

check assertFunction0 {
        { all arg2:Int,arg3:Int,arg1:Int | some arg2':Int | ((andGate[((arg1 >= arg3) => True else
False), ((arg3 >= 0) => True else False)]) in True)  => predFunction0[arg2,arg2',arg3,arg1] }
}

pred predFunction1 [arg2,arg2',arg2'':Int,arg3:Int,arg1,arg1':Int] {
       ((True) in True) and arg1'=funFunction0[arg2,arg2',1,arg1]  and arg2''=arg1'.add[arg3]
       ((((arg2'' = arg1.add[arg3].sub[1]) => True else False)) in True)                    // post
condition
}

fun funFunction1 [arg2,arg2',arg2'':Int,arg3:Int,arg1,arg1':Int] : Int {
       { return : Int | ((True) in True) and arg1'=funFunction0[arg2,arg2',1,arg1]  and
arg2''=arg1'.add[arg3] and return = arg2''}
}

check assertFunction1 {
        { all arg2:Int,arg3:Int,arg1:Int | some arg2':Int,arg2'':Int,arg1':Int | ((((arg3 > 1) => True
else False)) in True)  => predFunction1[arg2,arg2',arg2'',arg3,arg1,arg1'] }
}
```

Figure 29: Example Output – Functions

- Variable Declarations
- Arithmetic Operations
- Relational Operations
- Logical Operations
- Old syntax
- Functions, Function Declarations, Function Calls, Function Contracts
    - Please notice that one of the functions is calling another function in this code snippet.

- This program is logically correct.
- The output program compiles and verifies with no counterexamples in Alloy.

3.5 Example Input – Function with an If Statement

```
// Alyx got 7 on her English test.
// Marking Scheme 6-7 A 3-5 B 0-2 C
// A -> 1
// B -> 2
// C -> 3

fun int mark (int score, int result) {
    fun_require((score >= 0) && (score <= 7))
        if ((score >= 6) && (score <= 7)) {
            result = 1;
        } else if ((score >= 3) && (score <= 5)) {
            result = 2;
        } else {
            result = 3;
        }
        return result;
    fun_ensure((result >= 1) && (result <= 3))
}

int alyxScore = 7;
int alyxMark = mark(alyxScore, 0);
```

Figure 30: Example Input – Function with an If Statement

```
open logicFuncs

pred predFunction0 [arg1,arg1',arg1'',arg1''':Int,arg2:Int] {
	((True) in True) and  ( ((andGate[((arg2 >= 6) => True else False), ((arg2 <= 7) => True else
False)]) in True) =>
		((True) in True) and arg1'=1 and arg1'''=arg1'
	else (( andGate[((arg2 >= 3) => True else False), ((arg2 <= 5) => True else False)] ) in True)
=>
		((True) in True) and arg1''=2 and arg1'''=arg1''
	else
		((True) in True) and arg1'''=3 )
	((andGate[((arg1''' >= 1) => True else False), ((arg1''' <= 3) => True else False)]) in True)
			// post condition
}

fun funFunction0 [arg1,arg1',arg1'',arg1''':Int,arg2:Int] : Int {
	{ return : Int | ((True) in True) and  ( ((andGate[((arg2 >= 6) => True else False), ((arg2 <=
7) => True else False)]) in True) =>
		((True) in True) and arg1'=1 and arg1'''=arg1'
	else (( andGate[((arg2 >= 3) => True else False), ((arg2 <= 5) => True else False)] ) in True)
=>
		((True) in True) and arg1''=2 and arg1'''=arg1''
	else
		((True) in True) and arg1'''=3 )  and return = arg1'''}
}

check assertFunction0 {
	 { all arg1:Int,arg2:Int | some arg1':Int,arg1'':Int,arg1''':Int | ((andGate[((arg2 >= 0) =>
True else False), ((arg2 <= 7) => True else False)]) in True)  =>
predFunction0[arg1,arg1',arg1'',arg1''',arg2] }
}
```

Figure 30: Example Output – Function with an If Statement

- Variable Declarations
- Arithmetic Operations
- Relational Operations
- Logical Operations
- Functions, Function Declarations, Function Calls, Function Contracts
- If Statements with Else If
    - Notice that this time the if statement is in the function.

- This program is logically correct.
- The output program compiles and verifies with no counterexamples in Alloy.

3.6 Example Input - Loop

```
// Joan started learning adding numbers today!
// Will you help her on her adventure adding all numbers between 1-3?

int sum; // Joan will add each number to sum.
int current; // This is the current number Joan is looking at.
loop_require(true)
loop_init{
       sum = 0; // Joan has nothing added to sum yet. So she sets it to 0.
    current = 3; // Joan sets the current number before starting.
}
loop(current > 0 ) {
loop_invariant((sum == (((3-current) * ((3-current) + 1)) / 2)) && (current >= 0) && (current < 4) )
loop_variant(current)
       sum = sum + (4-current);
       current = current - 1;
}
loop_ensure(sum == 6)

// current sum in loop invariant is calculated using the formula n * (n + 1) / 2
```

Figure 31: Example Input – Loops

```
open logicFuncs

pred predForStatement0 [arg2,arg2',arg2'':Int,arg1,arg1',arg1'':Int] {
       (((True) in True) and arg1'=0 and arg2'=3
       =>
       ( andGate[andGate[((arg1' = 3.sub[arg2'].mul[3.sub[arg2'].add[1]].div[2]) => True else False),
((arg2' >= 0) => True else False)], ((arg2' < 4) => True else False)] in True ))

       (( andGate[andGate[((arg1 = 3.sub[arg2].mul[3.sub[arg2].add[1]].div[2]) => True else False),
((arg2 >= 0) => True else False)], ((arg2 < 4) => True else False)] in True )
        and( ((arg2 > 0) => True else False) in True ))
       =>
       ((True) in True) and arg2''=arg2.sub[1] and arg1''=arg1.add[4.sub[arg2]]
       =>
       (andGate[andGate[((arg1'' = 3.sub[arg2''].mul[3.sub[arg2''].add[1]].div[2]) => True else False),
((arg2'' >= 0) => True else False)], ((arg2'' < 4) => True else False)] in True and  ( arg2'' >= 0 )
and ( arg2 > arg2''))

       (( andGate[andGate[((arg1 = 3.sub[arg2].mul[3.sub[arg2].add[1]].div[2]) => True else False),
((arg2 >= 0) => True else False)], ((arg2 < 4) => True else False)] in True ) and not(( ((arg2 > 0) =>
True else False) in True ))
        =>
       ((((arg1 = 6) => True else False)) in True) )                    // post condition
}

check assertForStatement0 {
        { all arg2:Int,arg1:Int | some arg2':Int,arg2'':Int,arg1':Int,arg1'':Int | ((True) in True)  =>
predForStatement0[arg2,arg2',arg2'',arg1,arg1',arg1''] }
}
```

Figure 32: Example Output – Loops

- Variable Declarations
- Arithmetic Operations
- Relational Operations
- Logical Operations
- Loops

- This program is logically correct.
- The output program compiles and verifies with no counterexamples in Alloy.

3.7 Example Input - Loop

```
// Lucienne is trying to demonstrate her friend Alienor that and'ing any number of falses will still
yield false.
// Since Lucienne is a coder she decides to use functions to make it look fancy. (We have no idea
why Lucienne would ever want to do this using this compiler!)

fun bool andFalse (bool input, bool result) {
    fun_require(true)
    result = input && false;
    return result;
    fun_ensure(result <=> false)
}

int current;
bool result = true;

loop_require(result <=> true)
loop_init {
    current = 3;
}
loop(current > 0) {
    loop_invariant((current >= 0) && ( (current<3) => (result <=> false) ))
    loop_variant(current)
    result = andFalse(result, false);
    current = current - 1;
}
loop_ensure(result <=> false)
```

Figure 33: Example Input – Loops

```
open logicFuncs

pred predFunction0 [arg1,arg1':Bool,arg2:Bool] {
       ((True) in True) and arg1'=andGate[arg2, False]
       ((((arg1' in False) => True else False)) in True)                  // post condition
}

fun funFunction0 [arg1,arg1':Bool,arg2:Bool] : Bool {
       { return : Bool | ((True) in True) and arg1'=andGate[arg2, False] and return = arg1'}
}

check assertFunction0 {
        { all arg1:Bool,arg2:Bool | some arg1':Bool | ((True) in True)  =>
predFunction0[arg1,arg1',arg2] }
}

pred predForStatement0 [arg1,arg1':Bool,arg2,arg2',arg2'':Int,arg3,arg3':Bool] {
       (((True) in True) and arg3=False and arg2'=3
       =>
       ( andGate[((arg2' >= 0) => True else False), ((((((arg2' < 3) => True else False)) in True) =>
((((arg1 in False) => True else False)) in True)) => True else False)] in True ))

       (( andGate[((arg2 >= 0) => True else False), ((((((arg2 < 3) => True else False)) in True) =>
((((arg1 in False) => True else False)) in True)) => True else False)] in True )
        and( ((arg2 > 0) => True else False) in True ))
       =>
       ((True) in True) and arg1'=funFunction0[arg3,arg3',arg1]  and arg2''=arg2.sub[1] and arg3=False
       =>
       (andGate[((arg2'' >= 0) => True else False), ((((((arg2'' < 3) => True else False)) in True) =>
((((arg1' in False) => True else False)) in True)) => True else False)] in True and  ( arg2'' >= 0 )
and ( arg2 > arg2''))

       (( andGate[((arg2 >= 0) => True else False), ((((((arg2 < 3) => True else False)) in True) =>
((((arg1 in False) => True else False)) in True)) => True else False)] in True ) and not(( ((arg2 > 0)
=> True else False) in True ))
        =>
       ((((arg1 in False) => True else False)) in True) )                 // post condition
}

check assertForStatement0 {
        { all arg1:Bool,arg2:Int,arg3:Bool | some arg1':Bool,arg2':Int,arg2'':Int,arg3':Bool | ((((arg1
in True) => True else False)) in True)  => predForStatement0[arg1,arg1',arg2,arg2',arg2'',arg3,arg3'] }
}
```

Figure 34: Example Output – Loops

- Variable Declarations
- Arithmetic Operations
- Relational Operations
- Logical Operations
- Loops

- This program is logically correct.
- The output program compiles and verifies with no counterexamples in Alloy.

3.8 Example Input – Lists and Quantification Operations

```
// Bank Account Balance Monitor
// A Multinational bank wants to figure out if there are any empty accounts, but they would like to
know
// if there are multiple ways of doing it.
Int[] accountBalances;
accountBalances.add(400);
accountBalances.add(30000);
accountBalances.add(5000);
accountBalances.add(10000000);
accountBalances.add(0);

// First way with the existential quantifier.
Bool check1 = false;
if_require(true)
if(accountBalances.some(each == 0)) {
    check1 = true;
}
if_ensure(true)

bool check2 = false;
if_require(true)
if(accountBalances.some(each == 0)) {
    check2 = true;
}
if_ensure(true)
```

Figure 35: Example Input – Lists and Quantification Operations

```
open logicFuncs

pred predIfStatement0 [arg2,arg2':Bool,arg1:seq Int] {
      ((((some arrayElems: arg1.elems | arrayElems  in {each: Int | (((each = 0) => True else False)
in True)})=> True else False)) in True) =>
              ((True) in True) and arg2'=True
      ((True) in True)                 // post condition
}

check assertIfStatement0 {
       { all arg2:Bool,arg1:seq Int | some arg2':Bool | ((True) in True)  =>
predIfStatement0[arg2,arg2',arg1] }
}

pred predIfStatement1 [arg2,arg2':Bool,arg1:seq Int] {
      ((((some arrayElems: arg1.elems | arrayElems  in {each: Int | (((each = 0) => True else False)
in True)})=> True else False)) in True) =>
              ((True) in True) and arg2'=True
      ((True) in True)                 // post condition
}

check assertIfStatement1 {
       { all arg2:Bool,arg1:seq Int | some arg2':Bool | ((True) in True)  =>
predIfStatement1[arg2,arg2',arg1] }
}
```

Figure 36: Example Output – Lists and Quantification Operations

- Variable Declarations
- Arithmetic Operations
- Relational Operations
- Logical Operations
- If Statements with Contracts
- Lists, List Operations (Add), Quantification Operation

- This program is logically correct.

- The output program compiles and verifies with no counterexamples in Alloy.

3.9 Example Input – Function Calling Loop

```
// Gregory writes an algorithm that counts the number of integers that are a multiple of 3 and lower
than 100;
// Since the numbers are greater than 7 therefore Alloy won't behave normally (as explained in the
User Manual)

int counter = 0;

fun bool shouldIncrementCounter (int number, bool result) {
    fun_require((number >= 0) && (number < 100))
    result = false;
    if ((number % 3) == 0) {
        result = true;
    }
    return result;
    fun_ensure(true)
}

int current;
bool shouldIncrement;
loop_require(true)
loop_init {
    current = 99;
}
loop(current !=0 ) {
loop_invariant((current >= 0) && (current < 100) && (counter == ((99-current)/3)) )
loop_variant(current)
       shouldIncrement = shouldIncrementCounter(current, false);
       if (shouldIncrement ⇔ true) {
              counter = counter + 1;
       }
    current = current – 1;
}
loop_ensure(counter == 33)
```

Figure 37: Example Input – Function Calling Loop

```
open logicFuncs

pred predFunction0 [arg1,arg1’,arg1’’:Bool,arg2:Int] {
       ((True) in True) and arg1’=False and  ( ((((arg2.rem[3] = 0) => True else False)) in True) =>
              ((True) in True) and arg1’’=True )
       ((True) in True)                  // post condition
}

fun funFunction0 [arg1,arg1’,arg1’’:Bool,arg2:Int] : Bool {
       { return : Bool | ((True) in True) and arg1’=False and  ( ((((arg2.rem[3] = 0) => True else
False)) in True) =>
              ((True) in True) and arg1’’=True )  and return = arg1’’}
}

check assertFunction0 {
        { all arg1:Bool,arg2:Int | some arg1’:Bool,arg1’’:Bool | ((andGate[((arg2 >= 0) => True else
False), ((arg2 < 100) => True else False)]) in True)  => predFunction0[arg1,arg1’,arg1’’,arg2] }
}

pred predForStatement0 [arg1,arg1’,arg1’’:Int,arg3,arg3’:Bool,arg2,arg2’:Int,arg4,arg4’,arg4’’:Bool] {
       (((True) in True) and arg1’=99 and arg4=False
       =>
       ( andGate[andGate[((arg1’ >= 0) => True else False), ((arg1’ < 100) => True else False)], ((arg2
= 99.sub[arg1’].div[3]) => True else False)] in True ))

       (( andGate[andGate[((arg1 >= 0) => True else False), ((arg1 < 100) => True else False)], ((arg2
= 99.sub[arg1].div[3]) => True else False)] in True )
        and( ((arg1 != 0) => True else False) in True ))
       =>
       ((True) in True) and  ( ((((arg3’ in True) => True else False)) in True) =>
              ((True) in True) and arg4=False and arg2’=arg2.add[1] )  and arg4=False and
arg3’=funFunction0[arg4,arg4’,arg4’’,arg1]  and arg1’’=arg1.sub[1]
       =>
       (andGate[andGate[((arg1’’ >= 0) => True else False), ((arg1’’ < 100) => True else False)],
((arg2’ = 99.sub[arg1’’].div[3]) => True else False)] in True and  ( arg1’’ >= 0 ) and ( arg1 >
arg1’’))

       (( andGate[andGate[((arg1 >= 0) => True else False), ((arg1 < 100) => True else False)], ((arg2
= 99.sub[arg1].div[3]) => True else False)] in True ) and not(( ((arg1 != 0) => True else False) in
True ))
        =>
       ((((arg2 = 33) => True else False)) in True) )                    // post condition
}

check assertForStatement0 {
        { all arg1:Int,arg3:Bool,arg2:Int,arg4:Bool | some
arg1’:Int,arg1’’:Int,arg3’:Bool,arg2’:Int,arg4’:Bool,arg4’’:Bool | ((True) in True)  =>
predForStatement0[arg1,arg1’,arg1’’,arg3,arg3’,arg2,arg2’,arg4,arg4’,arg4’’] }
}
```

Figure 38: Example Output – Function Calling Loop

- Variable Declarations
- Arithmetic Operations
- Relational Operations
- Logical Operations
- If Statements with Contracts
- Functions, Function Declarations, Function Calls, Function Contracts
- Loops
- Please notice that loop is making a call to another function.

- This program is logically correct.

- The output program compiles and verifies with no counterexamples in Alloy. However, it is likely that Alloy is not behaving normally due to an issue involving bit-width. This will be explained in the “Limitations” chapter.

3.10 Example Input – Chef's Special

```
// Ege decided to write a simple list application to keep track of his marks at school.
// Causes state explosion (Skolemize error as explained in the User Manual)
int[] myMarks;

// Ege coincidentally got 70 on all his midterm tests, so he adds those first.
Int current;
loop_require(true)
loop_init {
    current = 3;
}
loop(current > 0) {
loop_invariant((current >= 0) && (current < 4) )
loop_variant(current)
    myMarks.add(70);
    current = current – 1;
}
loop_ensure(current == 0)

// Then he gets 35 and 90 on two tests.
myMarks.add(35);
myMarks.add(90);

// Then his prof says one of his quizzes can count for marks if he has any marks lower than 50 or
greater than 90.
// His prof also drops his lowest mark.
// On this quiz Ege got 80.

Bool hasLowMark = (myMarks.some(each < 50) || myMarks.some(each > 90));
if (hasLowMark ⇔ true) {
    myMarks.add(80);
    myMarks.remove(35);
}
```

Figure 39: Example Input – Chef's Special

```
open logicFuncs

pred predForStatement0 [arg1,arg1’,arg1’’:Int,arg2,arg2’:seq Int] {
       (((True) in True) and arg1’=3
       =>
       ( andGate[((arg1’ >= 0) => True else False), ((arg1’ < 4) => True else False)] in True ))

       (( andGate[((arg1 >= 0) => True else False), ((arg1 < 4) => True else False)] in True )
        and( ((arg1 > 0) => True else False) in True ))
       =>
       ((True) in True) and arg2’=arg2.add[70] and arg1’’=arg1.sub[1]
       =>
       (andGate[((arg1’’ >= 0) => True else False), ((arg1’’ < 4) => True else False)] in True and
( arg1’’ >= 0 ) and ( arg1 > arg1’’))

       (( andGate[((arg1 >= 0) => True else False), ((arg1 < 4) => True else False)] in True ) and
not(( ((arg1 > 0) => True else False) in True ))
        =>
       ((((arg1 = 0) => True else False)) in True) )                    // post condition
}

check assertForStatement0 {
       { all arg1:Int,arg2:seq Int | some arg1’:Int,arg1’’:Int,arg2’:seq Int | ((True) in True)  =>
predForStatement0[arg1,arg1’,arg1’’,arg2,arg2’] }
}
```

Figure 40: Example Output – Chef’s Special

- Variable Declarations
- Arithmetic Operations
- Relational Operations
- Logical Operations
- If Statements with Contracts
- Lists, List Operations (Add, Remove), Quantification Operation
- Loops

- This is just an ordinary example attempting to put together everything we did.

- This program is logically correct.
- The output program compiles but does not verify due to state explosion. It fails with the below error. This is mentioned in “Limitations”.
  “Analysis cannot be performed since it requires higher-order quantification that could not be skolemized.”

3.11 Example Input – Incorrect 3.1

```
// THIS IS ALMOST THE SAME FILE AS input-1.txt, BUT WITH THE OPERATORS FLIPPED IN THE
// IF BODY CAUSING POSTCONDITION TO FAIL ON PURPOSE
// ALLOY FINDS A COUNTER EXAMPLE!


// Bank Transaction Application

int cust1Balance = 100;
int cust2Balance = 600;
int cust3Balance = 300;

// customer 3 transfers funds to customer 1
cust3Balance = cust3Balance - 200;
cust1Balance = cust1Balance + 200;

// cust3Balance deposits 300
cust3Balance = cust3Balance + 300;

// cust1Balance withdraws 200
cust1Balance = cust1Balance - 200;

// cust3 asks cust1 for a loan of 200
// this time since customer 1 does not know how much money he has, he says he will loan the money if
he has enough.
int borrowAmount = 200;
if_require(cust1Balance > borrowAmount)
if (cust1Balance > borrowAmount) {
    cust1Balance = cust1Balance + borrowAmount;
    cust3Balance = cust3Balance - borrowAmount;
}
if_ensure((cust1Balance == cust1Balance_old - borrowAmount) && (cust3Balance == cust3Balance_old +
borrowAmount))
```

Figure 41: Example Input – Incorrect 3.1

```
open logicFuncs

pred predIfStatement0 [arg2:Int,arg1,arg1':Int,arg3,arg3':Int] {
       ((((arg3 > arg2) => True else False)) in True) =>
              ((True) in True) and arg1'=arg1.sub[arg2] and arg3'=arg3.add[arg2]
       ((andGate[((arg3' = arg3.sub[arg2]) => True else False), ((arg1' = arg1.add[arg2]) => True else
False)]) in True)                // post condition
}

check assertIfStatement0 {
        { all arg2:Int,arg1:Int,arg3:Int | some arg1':Int,arg3':Int | ((((arg3 > arg2) => True else
False)) in True)  => predIfStatement0[arg2,arg1,arg1',arg3,arg3'] }
}
```

Figure 42: Example Output – Incorrect 3.1

- This is just an incorrect version of 3.1.
- This program is logically incorrect.
- The output program compiles, and verification fails with a counterexample.

3.12 Example Input – Incorrect 3.7

```
// THIS IS ALMOST THE SAME FILE AS input-7.txt, BUT WITH THE ANDFALSE FUNCTION AND'ING TRUE INSTEAD
BY MISTAKE!
// ALLOY FINDS A COUNTER EXAMPLE!
// Lucienne is trying to demonstrate her friend Alienor that and'ing any number of falses will still
yield false.
// Since Lucienne is a coder she decides to use functions to make it look fancy. (We have no idea
why Lucienne would ever want to do this using this compiler!)

fun bool andFalse (bool input, bool result) {
    fun_require(true)
    result = input && true;
    return result;
    fun_ensure(result <=> false)
}

int current;
bool result = true;

loop_require(result <=> true)
loop_init {
    current = 3;
}
loop(current > 0) {
    loop_invariant((current >= 0) && ( (current<3) => (result <=> false) ))
    loop_variant(current)
    result = andFalse(result, false);
    current = current - 1;
}
loop_ensure(result <=> false)
```

Figure 43: Example Input – Incorrect 3.7

```
open logicFuncs

pred predFunction0 [arg1,arg1':Bool,arg2:Bool] {
       ((True) in True) and arg1'=andGate[arg2, True]
       ((((arg1' in False) => True else False)) in True)                  // post condition
}

fun funFunction0 [arg1,arg1':Bool,arg2:Bool] : Bool {
       { return : Bool | ((True) in True) and arg1'=andGate[arg2, True] and return = arg1'}
}

check assertFunction0 {
        { all arg1:Bool,arg2:Bool | some arg1':Bool | ((True) in True)  =>
predFunction0[arg1,arg1',arg2] }
}

pred predForStatement0 [arg1,arg1':Bool,arg2,arg2',arg2'':Int,arg3,arg3':Bool] {
       (((True) in True) and arg3=False and arg2'=3
       =>
       ( andGate[((arg2' >= 0) => True else False), ((((((arg2' < 3) => True else False)) in True) =>
((((arg1 in False) => True else False)) in True)) => True else False)] in True ))

       (( andGate[((arg2 >= 0) => True else False), ((((((arg2 < 3) => True else False)) in True) =>
((((arg1 in False) => True else False)) in True)) => True else False)] in True )
        and( ((arg2 > 0) => True else False) in True ))
       =>
       ((True) in True) and arg1'=funFunction0[arg3,arg3',arg1]  and arg2''=arg2.sub[1] and arg3=False
       =>
       (andGate[((arg2'' >= 0) => True else False), ((((((arg2'' < 3) => True else False)) in True) =>
((((arg1' in False) => True else False)) in True)) => True else False)] in True and  ( arg2'' >= 0 )
and ( arg2 > arg2''))

       (( andGate[((arg2 >= 0) => True else False), ((((((arg2 < 3) => True else False)) in True) =>
((((arg1 in False) => True else False)) in True)) => True else False)] in True ) and not(( ((arg2 > 0)
=> True else False) in True ))
        =>
       ((((arg1 in False) => True else False)) in True) )                 // post condition
}

check assertForStatement0 {
        { all arg1:Bool,arg2:Int,arg3:Bool | some arg1':Bool,arg2':Int,arg2'':Int,arg3':Bool | ((((arg1
in True) => True else False)) in True)  => predForStatement0[arg1,arg1',arg2,arg2',arg2'',arg3,arg3'] }
}
```

Figure 44: Example Output – Incorrect 3.7

- This is just an incorrect version of 3.7.
- This program is logically incorrect.
- The output program compiles, and verification fails with a counterexample.

3.13 Example Input – Incorrect 3.10

```
// THIS IS ALMOST THE SAME FILE AS input-10.txt, BUT NOW loop_ensure CHECKS FOR current IS EQUAL TO
2 WHICH IS INCORRECT!
// Line 16 is commented so that there won't be any skolemization issues with Alloy (as it is
explained in the user manual)
// ALLOY FINDS A COUNTER EXAMPLE!
// Ege decided to write a simple list application to keep track of his marks at school.
// Causes state explosion

int[] myMarks;

// Ege coincidentally got 70 on all his midterm tests, so he adds those first.
int current;
loop_require(true)
loop_init {
    current = 3;
}
loop(current > 0) {
loop_invariant((current >= 0) && (current < 4) )
loop_variant(current)
//    myMarks.add(70);
    current = current - 1;
}
loop_ensure(current == 2)

// Then he gets 35 and 90 on two tests.
myMarks.add(35);
myMarks.add(90);

// Then his prof says one of his quizzes can count for marks if he has any marks lower than 50 or
greater than 90.
// His prof also drops his lowest mark.
// On this quiz Ege got 80.

bool hasLowMark = (myMarks.some(each < 50) || myMarks.some(each > 90));
if (hasLowMark <=> true) {
    myMarks.add(80);
    myMarks.remove(35);
```

Figure 45: Example Input – Incorrect 3.10

```
open logicFuncs

pred predForStatement0 [arg1,arg1',arg1'':Int] {
       (((True) in True) and arg1'=3
       =>
       ( andGate[((arg1' >= 0) => True else False), ((arg1' < 4) => True else False)] in True ))

       (( andGate[((arg1 >= 0) => True else False), ((arg1 < 4) => True else False)] in True )
        and( ((arg1 > 0) => True else False) in True ))
       =>
       ((True) in True) and arg1''=arg1.sub[1]
       =>
       (andGate[((arg1'' >= 0) => True else False), ((arg1'' < 4) => True else False)] in True and
( arg1'' >= 0 ) and ( arg1 > arg1''))

       (( andGate[((arg1 >= 0) => True else False), ((arg1 < 4) => True else False)] in True ) and
not(( ((arg1 > 0) => True else False) in True ))
        =>
       ((((arg1 = 2) => True else False)) in True) )                    // post condition
}

check assertForStatement0 {
       { all arg1:Int | some arg1':Int,arg1'':Int | ((True) in True)  =>
predForStatement0[arg1,arg1',arg1''] }
}
```

Figure 46: Example Input – Incorrect 3.10

- This is just an incorrect version of 3.10.
- This program is logically incorrect.
- The output program compiles, and verification fails with a counterexample.

4 Miscellaneous Features

In this section we will be talking about the miscellaneous features we have in our compiler.

4.1 Type Checking

As mentioned earlier and demonstrated in section 1.1.3, our compiler also supports basic type checking. This is something trivial to do using couple of if statements since we are only supporting int and bool primitive data types and as well as lists. Another type that is possible to have is null for variables that are uninitialized. Nonetheless, type checking can be done very easily using couple of if statements since there a few possible cases are only. (i.e. few possible types: primitive, list or null)

Although this type checking can be done by implementing a separate visitor class just for that purpose, we are not doing that as similar functionality can be achieved by doing the type checking as we are parsing the input language into instances of objects in AntlrToInstruction class.

4.2 Error Reporting

Any semantic errors such as trying to assign an Integer value to a Boolean variable etc. will also be detected by our compiler and these semantic errors will be added to an internal data structure to be printed for user's attention.

4.3 Allowing Explicit Boolean Values in Alloy

As we have already mentioned, Alloy does not support explicit Boolean values. However, since it is common to have explicit Boolean values in our input language, we needed to figure out a way to support them. That's why we have a separate utilities library (imported by "open logicFuncs") that implements our version of True and False values as well as Boolean operations. This way we can have explicit Boolean values in Alloy.

In other words, there is no way to have True or False Boolean constants in Alloy code without or implementation of Boolean values. Even trivially correct (1 == 1) or incorrect (1==2) statements cannot be used in Alloy to simulate a similar functionality!

Also, as a result of this, it is possible to see bunch of statements in our output, like below.

```
(1 = 2 => True else False) in True
```

This statement says that if 1 equal to 2 return a Boolean value that has the result. The part in the parenthesis returns False (Our implementation of Booleans) since 1=2 is trivially incorrect. Then the returned Boolean is converted into Alloy's type of Booleans using the "in True" statement. This basically checks whether the Boolean returned from the part inside the parenthesis is an instance of True and returns the answer in Alloy's implementation of Booleans. Note that this statement returns False.

5 Limitations

In this section we will briefly talk about all the limitations in our compiler that we are aware of.

- Due to the way we are handling Booleans in Alloy (as mentioned in 4.3), it is possible to have trivially true Boolean statements such as just True in our output. This might cause Alloy Analyzer to raise warnings and Alloy unfortunately does not execute if it encounters any warnings. Therefore, the user needs to disable this functionality by clicking on "Alloy Warnings: yes" in the Options menu in Alloy Analyzer.
- We do not support List operations such as add and remove inside any block that has contracts, due to possibility of state explosion. Program still compiles though.
- We only support adding Boolean or Integer constants to Lists. That's why variables cannot be used to add Booleans or Integers to Lists.
- We are aware of an issue causing state explosion when in some cases existential quantifier is used in a postcondition. Program still compiles though.
- We only support non-nested loops.
- We do not support declaration of variables inside loop or if-statement bodies. Any variable declarations that are required must be done before implanting these statements (like C89 Standard for C).
- We do not support declaration of variables inside functions. Any variable declarations that are required must be done in the parameters of the function and a placeholder value should be passed while calling the function.
- We do not support accessing global variables from inside of functions due to scoping. Only local variables can be accessed inside functions at the moment.
    - As a result of this, we also do not support accessing lists from functions including usage of quantification operations over lists. Doing so will cause a Null Pointer Exception. This exception could easily be replaced by an error message, but we will not be doing that due to timing constraints.
- We do not support returning directly arithmetic or logical operations such as x – y or x && y from functions. That's why, to be able to return results from functions, a separate variable must be declared, and this variable should be returned instead (Of course once the calculated result is assigned to it).
- We only support contracts in the parent statement in nested if statements. For example, if you have triple nested if statements, then only the first parent can have contracts.
- Here's also another issue we have come across regarding the way our compiler handles function calls.
  Passing the same variable in the place of more that one argument to a function might cause semantically incorrect output. Program still compiles though. We illustrate this issue below.
  Here's an example to illustrate this issue.

```
int fun1(int x, int y){
fun_require(true)
       x=x+y;
```

```
        return x;
fun_ensure(true)
}
```

Figure 46: Known Issue 1 – Function - Code Snippet 1

Internally the compiler translates the function in Figure 46 to something like in Figure 47 to be able to store the pre and post states of the variable x.

```
int fun1(int x, int x', int y){
fun_require(true)
        x' = x+y;
        return x';
fun_ensure(true)
}
```

Figure 47: Known Issue 1 – Function - Code Snippet 1 Translated Internally

```
int result;
int a = 4;
int b = 3;
result = fun1(a,b);
```

Figure 48: Known Issue 1 – Function Call - Code Snippet 2

```
int result;
int a = 4;
int a';
int b = 3;
result = fun1(a,a',b);
```

Figure 49: Known Issue 1 – Function Call - Code Snippet 2 Translated Internally

If the user calls the translated function in Figure 47 like in Figure 48, this causes the function call internally to be translated to something like in Figure 49 for the post state of variable a.

```
int result;
int a = 4;
int b = 3;
result = fun1(a,a);
```

Figure 50: Known Issue 1 – Function Call - Code Snippet 3

```
int result;
int a = 4;
int a';
int a'';
int b = 3;
result = fun1(a,a',a'');
```

Figure 51: Known Issue 1 – Function Call - Code Snippet 3 Translated Internally

```
int result;
int a = 4;
int a';
int b = 3;
result = fun1(a,a',a);
```

Figure 52: Known Issue 1 – Function Call - Code Snippet 3 Translated Internally (Expected)

But on the other hand, if the user calls the function like in Figure 50, then in that case, it will be internally translated to something like in Figure 51 whereas it should have translated into something like in Figure 52.

There are also some inconsistencies in Alloy Analyzer / Alloy we have come across that we believe it is worthwhile to mention here.

- Inconsistency 1:

  The code below in Figure 53 does not generate any counter examples whereas it should since the predicate ***pred predFunction0*** is not correct for all integers.

```
open logicFuncs

pred predFunction0 [] {
	3=3
}

check assertFunction0 {
	 predFunction0[]
}

pred predForStatement0 [arg1: Int] {
```

```
        (arg1 = 3)
}

check assertForStatement0 {
        { all arg1:Int | predForStatement0[arg1] }
}
```

Figure 53: Alloy Inconsistency 1 – Example 1

```
open logicFuncs

pred predFunction0 [] {
        3=3
}

//check assertFunction0 {
//      predFunction0[]
//}

pred predForStatement0 [arg1: Int] {
        (arg1 = 3)
}

check assertForStatement0 {
        { all arg1:Int | predForStatement0[arg1] }
}
```

Figure 54: Alloy Inconsistency 1 – Example 2

However, when we remove ***check predFunction0*** by commenting it out Alloy finds a counter example just like it should. We are not sure why this issue arises; it is most likely an Alloy issue.

- Inconsistency 2:

  Using large values for inputs causes inconsistencies with Alloy Analyzer / Alloy. Since the default bit-width for verification in Alloy is 4 bits, Alloy Analyzer will have issues when working with values that are not possible to represent using 4 bits, unless the user specifically changes the bit-width in Alloy. As a result, Alloy Analyzer will have issues when working with values out of the range [-8,7] (both inclusive).

- Inconsistency 3:

```
        open logicFuncs


        pred predIfStatement0 [arg1: Int, arg1': Int] {
              arg1'=3
```

```
}

pred predIfStatement1 [arg1: Int, arg1': Int] {
      arg1'=3 => arg1'=2
}

check assertIfStatement0 {
       { all arg1:Int | some arg1':Int |
predIfStatement0[arg1,arg1'] }
}

check assertIfStatement1 {
       { all arg1:Int | some arg1':Int |
predIfStatement1[arg1,arg1'] }
}
```

Figure 55: Alloy Inconsistency 3 – Example 1

- In the given example, check `assertIfStatement0` succeeds because assigning constants to arg1' does not generate any counterexamples. Based on our understanding, since arg1' is declared as a free variable (i.e. it is declared as some arg1':Int) therefore there exist a case where arg1'=3 is true.

- On the other hand, pred `predIfStatement1` does not generate counterexamples either. However, this is incorrect because the implication can not possibly be correct since `arg1'=3 => arg1'=2.` Meaning, the case where arg1'=3 is true arg1'=2 cannot be correct, and therefore this should generate counterexamples.

- Luckily this only causes issues when translating loops and does not affect the translation of other statements.