# Verification of an iterative implementation of Tarjan's algorithm for Strongly Connected Components using Dafny

Schols, Wouter R.M.

*Award date:*
2020


Department of Mathematics and Computer Science
Formal System Analysis Research Group

# Verification of an iterative implementation of Tarjan's algorithm for Strongly Connected Components using Dafny

*Master Thesis*

W.R.M. Schols

Supervisors:
dr.ir. J.W. Wesselink
dr. C. Huizing

version 1.0

Eindhoven, November 2020

# Abstract

In this paper we will present a formal proof of Tarjan's algorithm for strongly connected components using the automated program verifier Dafny. In this paper we introduce and verify two different implementations of Tarjan's algorithm, a recursive implementation and an iterative implementation. The recursive implementation is similar to the original algorithm introduced by R.E. Tarjan [14]. The iterative implementation improves on the recursive version by preventing stack overflow problems and is used in the mCRL2 toolset. In this paper we formally prove that both version of the algorithm are correct using Dafny. We prove correctness of the algorithm using proof annotations. These proof annotations are specified directly in terms of the program variables. The proof of Tarjan's algorithm for SCC is complex and hard to directly implement in Dafny. When proofs become too complex then Dafny will become slow and Dafny can exhibit unexpected behavior when failing to verify a proof. We introduce generic techniques which can be used to implement a complex proofs while reducing the complexity for Dafny. We then use these techniques to implement a stable proof of Tarjan's algorithm for SCC.

# Preface

I started planning my master thesis in September of 2019 and now in November 2020 I have finally finished my thesis. A lot of things have happened during these months. It all started in the summer of 2019 when I had a conversation with professor J.F. Groote. I wanted to find a graduation project abroad at a company which used high tech verification techniques. Through different professors at universities all over the world I finally landed at the perfect internship at Amazon's Automated Reasoning Group in Seattle. Because of the Corona virus and various other circumstances this became one of the most unique experiences I have ever had.

The internship was supposed to start January 2020, however my visa took a lot more time. My internship was moved to March because it took more time then expected to obtain a visum. In the end I received my visa on a Thursday, left on Friday and arrived in Seattle on Saturday. The Monday after I arrived the airports and the Amazon offices closed down because the Corona pandemic started. Because of the Corona epidemic I had to work from different hotel rooms. The content of the internship was largely unaffected but all meetings were moved online. Unfortunately, everything else was affected. Some of the biggest problems included the hotel I was staying shutting down unexpectedly, which forced me to move instantly, and all government offices shutting down preventing me from applying for vital tax information. Once my internship ended in June the Corona virus had calmed down somewhat but the riots started. Unfortunately I needed to take a bus from the center of Seattle in the middle of the night. Luckily there were no riots at that time. Overall even though the internship was very stressful at times the internship was a great experience and I learned a lot. I would have like to explore the area more and meet new people but it was not meant to be.

Once I finally arrived home the problems where not over. My internship was supposed to be part of a larger project at Amazon. This project should have become public in May but because of the Corona virus this project was delayed and did not go public. Because of this I could not write my master thesis about my internship. This is why I had to search for a new graduation subject once I arrived back home. I found that J.W. Wesselink and C. Huizing were working on a project which used the technologies which I also used during my internship. When I started working on the project it was even unknown if finishing the project was feasible. Using the experience and skills I gained during the internship I was able to quickly pick up the project and solve the open problems. In the end I was able to tackle all problems which resulted in this paper.

I want to especially thank prof. dr. ir. J.F. Groote for all the assistance to find and guide my internship, my internship mentor dr. K.R.M. Leino from Amazon for his guidance during the internship and my graduation supervisors dr. ir. J.W. Wesselink and dr. C. Huizing for their contributions to the paper.

# Contents

# List of Definitions

# Chapter 1

# Introduction

The mCRL2 toolset depends on various algorithms including the Tarjan's strongly connected component algorithm. The functional correctness of these algorithms is vital for the correctness of the mCRL2 toolset. The recursive version of Tarjan's strongly connected components algorithm has been verified using Dafny in an internal project at Eindhoven University of technology. [8] Unfortunately the recursive algorithm is not suitable for mCRL2 since the stack overflows in large graphs. This is why a more complex iterative version of Tarjan's strongly connected component algorithm is used [16].

We want to verify the correctness of this iterative version of the strongly connected components algorithm. Unfortunately the existing verification of the recursive algorithm is performance intensive and too inflexible. We could not trivially adapt the existing verification to handle the complexity of the iterative algorithm. Verifying the iterative version of Tarjan's algorithm for strongly connected components will be the main goal of this master thesis. We aim to answer the following question "*Can we verify the iterative version of Tarjan's algorithm for strongly connected components using Dafny?*" In order to solve the problem we want to answer the sub question "*How can we optimize proof to verify programs in Dafny?*".

In this paper we will present a formal proof of Tarjan's algorithm for strongly connected components using the automated program verifier Dafny. Henceforth, we refer to Tarjan's algorithm for strongly connected components as Tarjan's algorithm. In this paper we introduce and verify the two different implementations of Tarjan's algorithm, a recursive implementation and an iterative implementation. The recursive implementation is similar to the original algorithm introduced by R.E. Tarjan [14]. The iterative implementation improves on the recursive version by preventing stack overflow problems and is used in the mCRL2 toolset. In this paper we formally prove that both version of the algorithm are correct using Dafny. We prove correctness of the algorithm using proof annotations. These proof annotations are specified directly in terms of the program variables.

We face two major challenges when verifying Tarjan's algorithm. The first challenge is defining the proof annotations and invariants. We want to define proof annotations and invariants directly in terms of program variables. This makes it easier to understand the proof and to reuse partial proofs between the two versions of the algorithm. Defining invariants which are strong enough to prove that the algorithm is correct can be a challenge. The second problem is that the proof needs to be implemented in Dafny. Dafny is an automatic program verifier which means that Dafny uses a SAT solver to automatically prove that the algorithm meets its specification. Once programs get more complex we need to use special techniques to simplify the verification process to create a proof which Dafny can verify. In this paper we introduce techniques which can be used to implement complex proofs in Dafny.

It is recommended that the reader has experience using Dafny before reading this paper. Chapter 2 briefly introduces the Dafny language and the some underlying concepts. Both versions of Tarjan's algorithm are introduced in chapter 3. A Dafny proof which proves correctness of the recursive Tarjan's algorithm already existed internally at Eindhoven University of Technology.

This pre-existing proof is introduced in chapter 4. This proof faced a lot of performance issues because of structural problems in the implementation. Techniques to resolve these issues are introduced in chapter 6. In chapter 6 a new stable proof is introduced for the recursive version of Tarjan's algorithm. This new proof is based on the pre-existing proof. This recursive proof and the techniques to implement a stable proof can be reused in the proof of the iterative version of Tarjan's algorithm. The proof of the iterative algorithm is introduced in chapter 7.

# Chapter 2

# Preliminaries

In this paper we discuss how the Tarjan's algorithm for Strongly Connected Components can be verified in Dafny. Henceforward, we refer to the Tarjan's algorithm for Strongly Connected Components as the Tarjan's algorithm. It is recommended that the reader has some experience with Dafny before reading this paper. In this section we briefly introduce the Dafny language and the main ideas behind the language.

Dafny is an imperative language which supports formal specification. Dafny allows users to specify the expected behaviour of a method by defining preconditions and postconditions. Dafny will attempt to prove that the preconditions hold whenever a method is called and that the postconditions hold once a method has terminated. The Dafny code consists of two types of code, the true program code and the verification code. The true program code is code which can be compiled. The verification code cannot be executed but can be verified. When Dafny code is compiled then the Dafny verifier will use the theorem prover Z3 to prove that the verification code is correct. When the Dafny verifier succeeds then the compiler will remove all verification code and compiles the executable code. Dafny ensures us that the behaviour of the compiled code complies to the specified behaviour.

## 2.1 Basics of verification using Dafny

In Dafny methods have a specification which consists of a precondition and a postcondition. The user of Dafny has to manually create this specification for every method. A method contains a program which consists of a series of operations. Dafny already has a built-in specification for every operation. Dafny proves that the precondition of every operation holds before the operation is executed. For example, take the operation $z := x/y$. This operation has the implicit precondition that $y \neq 0$. As another example, the operation $z := array[i]$ accesses an array at index $i$. This operation has the precondition that the index $i$ is smaller the length of the array. Dafny verifies a method by proving that the postcondition holds at the end of a method and that the method terminates given that the preconditions hold at the start of the method.

As a concrete example take the Dafny code from listing 2.1. This code introduces two method `ABSIntegers` and `MakePositive`. The method `ABSIntegers` accepts an array of integers as input and returns an array with the absolute value of the integers as output. The behaviour of the `ABSIntegers` method is formally specified in lines 0 to 2 from listing 2.1. At line 0 it is specified that the method `ABSIntegers` accepts an array of integers as input and returns an array of natural numbers as output. The input array is called *numbers* and the output array is called *res*. By definition a natural number is a non negative integer. At line 1 and 2 it is specified that *numbers* and *res* have the same size and that for every index $i$ in *numbers* it holds that $numbers[i] = res[i]$ or $numbers[i] = -res[i]$. By definition all elements in *numbers* are positive so we know that for every $i$ it holds that $numbers[i] = \texttt{ABS}(res[i])$.

The method `ABSIntegers` calls a helper method `MakePositive`. This method converts an

integer *number* into a natural number *res*. As defined in line 24, this method has the postcondition that $res = -number$. If *number* is a positive integer then *res* is a negative natural number. Dafny will not allow this and return an error. This is why the method `MakePositive` requires the precondition that *number* is negative from line 23. Dafny can now prove that *number* is positive.

```
0  method ABSIntegers(numbers: array<int>) returns(res: array<nat>)
1    ensures numbers.Length = res.Length
2    ensures ∀ i | 0 ≤ i < numbers.Length  • numbers[i] = res[i] ∨ res[i] = −1 ∗ numbers[i]
3  {
4    var index := 0;
5    res := new nat[numbers.Length];
6    while index < numbers.Length
7      invariant 0 ≤ index ≤ numbers.Length
8      invariant numbers.Length = res.Length
9      invariant ∀ i | 0 ≤ i < index  • numbers[i] = res[i] ∨ res[i] = −1 ∗ numbers[i]
10   {
11     if numbers[index] < 0 {
12       var posNumber := MakePositive(numbers[index]);
13       res[index] := posNumber;
14     }else{
15       res[index] := numbers[index];
16     }
17     index := index + 1;
18   }
19   return res;
20 }
21
22 method MakePositive(number: int) returns(res: nat)
23   requires number ≤ 0
24   ensures −1 ∗ number = res
25 {
26   return −number;
27 }
```

Listing 2.1: Example Dafny code

Dafny verifies a method based on Hoare triples [7]. A Hoare triple $\{Q\}S;\{P\}$ states that if precondition $Q$ holds before program $S$ then operation $P$ holds after program $S$ given that program $S$ terminates. These Hoare triples can be combined. If Hoare triples $\{Q\}S_1;\{P\}$ and $\{P\}S_2;\{R\}$ hold then Hoare triple $\{Q\}S_1;S_2;\{R\}$ holds. Dafny proves that Hoare triples hold for different sections of code. These sections are then combined to prove that the if the precondition holds before the method then the postcondition holds after the method. Dafny can automatically derive the required Hoare triples for simple sections of programs. Unfortunately Dafny cannot automatically derive the Hoare triple of loops and method calls. The user is required to specify a loop invariant for every loop and a specification for every method. Given a loop **while** $B$ **do** $S$ **end** the Dafny user has to define a loop invariant $P$. Dafny has to prove that the operations $S$ maintains the loop invariant, i.e. $\{\neg B \wedge P\}S\{P\}$, in a separate proof. Dafny can then derive that the Hoare triple $\{P\}$ **while** $B$ **do** $S_1 \dots S_n$ **end** $\{P \wedge \neg B\}$ holds. Similarly a method $M$ with precondition $Q$ and postcondition $P$ is verified in a separate proof. Dafny can then derive a Hoare triple for the method call $\{Q\}M();\{P\}$.

Continuing the example from listing 2.1, Dafny cannot verify the method `ABSInteger` if the user does not define a loop invariant and a pre and postcondition of the method `MakePositive`. Once the user has defined these invariants then Dafny can easily prove that the postcondition holds. We want to define some loop invariants such that the postconditions from lines $1, 2$ can be derived from the loop invariant. Next to this the loop invariant should hold at the start of the loop at line 5. The loop invariants from lines 8 and 9 allow us to derive the postcondition of the method after the loop. In order to ensure that *index* is within the bounds of the array we also require the loop invariant from line 7. Dafny proves that the method is correct by verifying the Hoare triples.

$$\{true\}l_4;l_5;\{Pred_7 \wedge Pred_8 \wedge Pred_9\}$$

$$\{Pred_7 \wedge Pred_8 \wedge Pred_9 \wedge index < numbers.length\}l_{11};\dots;l_{17};\{Pred_7 \wedge Pred_8 \wedge Pred_9\}$$

$$\{Pred_7 \wedge Pred_8 \wedge Pred_9 \wedge index \geq numbers.length\}l_{19};\{Pred_1 \wedge Pred_2\}$$

Here $l_i$ is line $i$ of our example and $Pred_j$ is the invariant at line $j$.

The first and last Hoare triples are easy to derive. The Hoare triple which proves that the loop maintains the loop invariant is slightly harder to derive. Dafny needs to verify the method call at line 12 with the specification of the `MakePositive` method. This results in the Hoare triple for line 12:

$$\{numbers[index] \leq 0\}\ \texttt{MakePositive}\ \{-1 * numbers[index] = posNumber\}$$

Using this Hoare triple the proof that the loop maintains the loop invariant is trivial.

Once programs get more complex Dafny might fail to verify a method. Solving these verification issues can be challenging. Understanding how Dafny verifies methods is an important first step into solving the verification issues.

## 2.2 Deterministic and nondeterministic code

As discussed before Dafny code can be split into two different groups of code, the true program code and the verification code. The true program code is deterministic code which determines the run time behaviour of the program. The verification code is nondeterministic code which specifies the desired behaviour of the program. New Dafny users might be confused by this distinction. A new Dafny user will discover that some expressions are allowed in some locations but return errors in other locations. A Dafny user should be aware if the code is deterministic or nondeterministic. Deterministic code is always located inside a method. All code which calls other methods or which modifies a variable not labeled as *ghost* is part of the true program code and therefore deterministic. All other code: preconditions, postconditions, lemmas, ghost variables, assert statements and functions are part of the verification code.

Although it is often desirable Dafny does not allow verification code to call methods. Take the example from listing 2.1. We are not allowed to replace the postcondition from line 2 with the postcondition:

$$\forall i | 0 \leq numbers.Length \bullet res[i] = numbers[i] \vee res[i] = \texttt{MakePositive}(numbers[i])$$

However specifying a pre or postcondition by calling another method might be desirable. In these cases one can define a function with the same behaviour as the method. In the example we could define the function $f(x : \textbf{int}) : \textbf{nat} = -x$, with the precondition that $0 \leq x$. Functions can be called from the specification code and one does not have to define a postcondition for a function. The postcondition of line 2 could be replaced by:

$$\forall i | 0 \leq numbers.Length \bullet res[i] = numbers[i] \vee res[i] = f(numbers[i])$$

The postcondition of line 24 should then be replaced by $res = f(number)$. Functions are a vital tool to specify complex behaviour, however, they can cause a lot of confusion for new Dafny users. Since functions are part of the verification code it follows that functions can be nondeterministic. This means that the function body can contain nondeterministic expressions. Note that functions are still a congruence so a function will always return the same value for the same input, even if the function body in nondeterministic. The common property that if $x = y$ then $f(x) = f(y)$ still holds even if the function does not have a deterministic specification.

A second common source of confusion are the *let* expressions. The expression $\textbf{var}\ x : X : |P$ states that $x$ is a variable of type $X$ such that the predicate $P$ holds. This expression is commonly used in Dafny programs. When a *let* expression is used in the verification code then Dafny has to prove that an element satisfying the predicate exists. If a *let* expression is used in the true program code then one also has to prove that there is only one value of $x$ which satisfies the predicate $P$.

# Chapter 3

# Tarjan's algorithm for strongly connected components

In this paper we verify a recursive and an iterative implementation of Tarjan's algorithm for strongly connected components. The recursive implementation of Tarjan's algorithm is very close to the original version from the paper "*Depth-first search and linear graph algorithms*" by R.Tarjan [14]. The recursive algorithm has a downside that the recursion stack can overflow in large graphs. This is why J. Öqvist proposed an iterative implementation in "*Iterative Tarjan strongly connected components in python*" [16]. The iterative implementation of Tarjan's algorithm prevents any stack overflow problems and is used in the mCRL2 toolset [15].

## 3.1 Fundamental idea of Tarjan's algorithm

The Tarjan's algorithm accepts a graph as input and calculates all strongly connected components in the graph. A strongly connected component or SCC is a maximal set of nodes $C$ such that all nodes in $C$ are connected. An exact definition of *connected* is given in definition 3.3 and a definition of strongly connected components is given in definition 3.4. Note that by definition of *connected* a node is always connected to itself.

Both implementations of Tarjan's algorithm calculate the SCC's by calling the `StrongConnect` algorithm. The `StrongConnect` algorithm accepts a start node as input. The `StrongConnect` algorithm iterates over all nodes which are reachable from the start node and divides them into SCC's. The main Tarjan's algorithm calls the `StrongConnect` method until all nodes are part of an SCC. The main method of Tarjan's algorithm is identical in both versions of the algorithm. However, the implementation of the `StrongConnect` algorithm differs. The main algorithm can be found in algorithm 1, the recursive `StrongConnect` algorithm is algorithm 2 and the iterative `StrongConnect` algorithm is algorithm 3. Both implementations of the `StrongConnect` method use a function *successors*($u$). This function returns a sequence containing all successors of node $u$ where a successor is defined in definition 3.2.

The recursive `StrongConnect` algorithm, algorithm 2, creates components by assigning a number to all nodes in order of discovery. At the start of the algorithm, line 1 to line 4, the input node $u$ is added to the variables *stack*, *disc*, *low*. The variable *disc* is a mapping from nodes to natural numbers. This variables stores the discovery number of all discovered nodes. The variable *low* is also a mapping. The entry *low*[$u$] stores the lowest *disc* value of a node which is reachable from $u$ but not already part of a component. Since all nodes are reachable from itself the value of *low*[$u$] is initially *disc*[$u$]. The last variable *stack* contains all nodes with a *disc* value which are not part of a component. Note that if a node is located below $u$ in the *stack* then it has a lower *disc* value. Once node $u$ has been added to the global variables then the `StrongConnect` algorithm iterates over all successors of node $u$, line 5. The algorithm tries to lower *low*[$u$] by comparing it to a successor. If a successor of $u$ does not have a *disc* value then the successor still needs to be

processed. This is done using a recursive call at line 12. After this recursive call the $low[u]$ value can be updated. Once all successors of $u$ have been processed then the algorithm checks if the value of $low[u]$ has decreased in line 11. If $low[u] < disc[u]$ then there is a node $v$ below $u$ in the stack which is in a cycle with $u$. Since $u$ and $v$ are in a cycle both nodes need to be in the same SCC. The algorithm leaves $u$ on the *stack* such that it can be removed later. If no node with a lower *disc* value has been found then $disc[u] = low[u]$. This implies that $u$ is not connected to any nodes before it in the *stack*. All nodes after $u$ in the stack are in a cycle with node $u$ so $u$ and all nodes above $u$ in the stack form a SCC. The loop at line 13 removes $u$ and all nodes above $u$ from the stack and creates an SCC. A proof that the algorithm is correct will be given in chapter 4.

The iterative implementation of `StrongConnect` follows the same principles. The difference between the two implementations is that the iterative algorithm manually pushes two local variables on a stack instead of using recursion. The order in which nodes are processed and variables are updated is the same in both versions.

**Definition 3.1** (Graph)**.** A directed graph $G(V, E)$ consists of a set of nodes $V$ and a set of edges $E$. Here $V$ is of the type $set\langle Node\rangle$ and an edge is a relation between nodes, $E : Node \times Node \to \mathbb{B}$.

**Definition 3.2** (Successors)**.** The function $successors(u : Node) : set\langle Node\rangle$ returns all successor of a node $u$.

$$successors(u) \equiv \{v \mid E(u, v)\}$$

**Definition 3.3** (Connected)**.** The predicate $Connected(u : Node, v : Node)$ holds if a node $u$ is connected to a node $v$. A node $u$ is connected to $v$ if and only if there is a sequence of nodes $p \equiv \langle p_0, \ldots p_{|p|-1}\rangle$ starting with $u$ and ending with $v$ where every node $p_i$ in $p$ is connected to its successors $p_{i+1} \in p$.

$$connected(u, v) \equiv \exists p : seq\langle Node\rangle :: p_0 = u \wedge p_{|p|-1} = v \wedge \forall i : 0 \leq i < |p| - 1 :: E(p_i, p_{i+1})$$

**Definition 3.4** (Strongly Connected Component)**.** A Strongly Connected Component or SCC is a set of nodes $C$ in a graph $G(V, E)$ such that $C \subseteq V$ and for all nodes $u, v \in C$ it holds that $u$ is connected to $v$ and there is no a strongly connected components $C'$ such that $C \subset C'$.

$$SCC(C) \equiv \forall u, v \in C :: connected(u, v) \wedge \forall C' : SCC(C') :: C \not\subset C'$$

**Algorithm 1** Tarjan's Strongly Connected Component Algorithm

**Input**: $G = (V, E)$: A graph with nodes $V$ and edges $E$.
**Output**: *result*: A sequence containing all strongly connected components of the graph.
TARJAN($G$):

```
1: var stack := [ ]
2: var low := {:}                    ▷ the empty mapping is denoted as {:}
3: var disc := {:}                   ▷ u ∈ low means u is a key of mapping low
4: var result := [ ]
5: for u ∈ V do
6:     if u ∉ low then
7:         StrongConnect(u)
8: return result
```

**Algorithm 2** Recursive version of the helper function StrongConnect

**Input**: $u$: An element of *Node*.
StrongConnect($u$):

```
1:  var k := |disc|                          ▷ k is the amount of nodes discovered before u
2:  var disc[u] := k
3:  var low[u] := k                          ▷ initially low[u] = disc[u]
4:  var stack := stack ++ [u]
5:  for v ∈ successors(u) do
6:      if v ∉ low then
7:          StrongConnect(v)
8:          low[u] := min(low[u], low[v])
9:      else if v ∈ stack then
10:         low[u] := min(low[u], disc[v])
11: if low[u] = disc[u] then                 ▷ a SCC has been found
12:     comp := [ ]
13:     while true do
14:         v := stack[|stack| − 1]          ▷ assign the top of the stack to v
15:         stack := stack[0 : |stack| − 1]   ▷ pop an element from the stack
16:         comp := comp ++ [v]
17:         if v = u then break
18:     result := result ++ [comp]
```

**Algorithm 3** Iterative version of helper function StrongConnect

**Input**: $u$: An element of *Node*.
StrongConnect($u_0$):

```
 1: work := [(u_0, 0)]
 2: while |work| > 0 do
 3:     (u, j) := work[|work| − 1]
 4:     var work := work[0 : |work| − 1]                ▷ pop an element from the work array
 5:     if j = 0 then
 6:         var k := |disc|                             ▷ k is the discovery time that is assigned to node u
 7:         disc[u] := k
 8:         low[u] := k                                 ▷ initially low[u] = disc[u]
 9:         stack := stack ++ [u]
10:     recurse := false
11:     for i := j to |successors(u)| − 1 do
12:         v := successors(u)[i]
13:         if v ∉ low then
14:             work := work ++ [(u, i + 1)]
15:             work := work ++ [(v, 0)]
16:             recurse := true
17:             break
18:         else if v ∈ stack then
19:             low[u] := min(low[u], disc[v])
20:     if ¬recurse then
21:         if low[u] = disc[u] then                    ▷ an SCC has been found
22:             comp := [ ]
23:             while true do
24:                 v := stack[|stack| − 1]             ▷ assign the top of the stack to v
25:                 stack := stack[0 : |stack| − 1]     ▷ pop an element from the stack
26:                 comp := comp ++ [v]
27:                 if v = u then break
28:             result := result ++ [comp]
29:         if |work| > 0 then
30:             v := u
31:             (u, j) := work[|work| − 1]
32:             low[u] := min(low[u], low[v])
```

## 3.2 An example of Tarjan's algorithm

In section 3.1 the ideas behind Tarjan's algorithm are introduced. These ideas can be hard to grasp. An example iteration of the algorithm is illustrated in figure 3.1. In this example we start at node $A$. Since all nodes are reachable from node $A$ the `StrongConnect` algorithm is only called once. First the node $A$ is added to the variables *stack*, *disc* and *low* the value of $disc[A]$ and $low[A]$ is set to 0, panel 2. After this a successor of node $A$ should be processed. We choose to process the successor $B$ first. Node $B$ is added to the *stack* and $disc[B]$ and $low[B]$ are set to 1, panel 3. Since node $B$ has no successors and since $disc[B] = low[B]$ node $B$ is removed from the *stack* and a new component $C_1 = \{B\}$ is created, panel 4.

After this we process the remaining successor of node $A$. Node $C$ is added to the *stack* and $disc[C]$ and $low[C]$ are set to 2, panel 5. Since node $C$ has a successor we process node $D$ next. Node $D$ is added to the *stack* and $disc[D]$ and $low[D]$ are set to 3, panel 6. Node $D$ is connected to node $E$ so node $E$ is added to the *stack* and $disc[E]$ and $low[E]$ are set to 4, panel 7. Since $C$ is a successor of $E$ and $C$ is below $E$ in the *stack* $low[E]$ is compared to $disc[C]$. Since $disc[C]$ is 2 and $low[E]$ is 4 the value of $low[E]$ is set to 2. We have now processed all neighbours of $E$. Since $low[E] < disc[E]$ we leave $E$ on the *stack*, panel 8. Next we continue processing node $D$. Since $low[D] < low[E]$ we set $low[D]$ to $low[E]$. We have now processed all neighbours of node $D$. Since $low[D] < disc[D]$ we leave $D$ on the *stack*, panel 9. Lastly we continue processing node $C$. Since $low[C] = low[D]$ the value of $low[C]$ remains unchanged. Since $low[C] = disc[C]$ node $C$ and all nodes above it are removed from the *stack* and added to a new component $C_2 = \{C, D, E\}$, panel 10. This leaves us with the first node $A$. Since $low[A] < disc[C]$ the value of $low[A]$ is not updated. Since $low[A] = disc[A]$ we create the last component $C_3 = \{A\}$. All nodes have now been processed and divided over SCC. Note that the variable *stack* is empty and all nodes connected to $A$ have been processed. As we will prove later this is always the case after any call of `StrongConnect`.

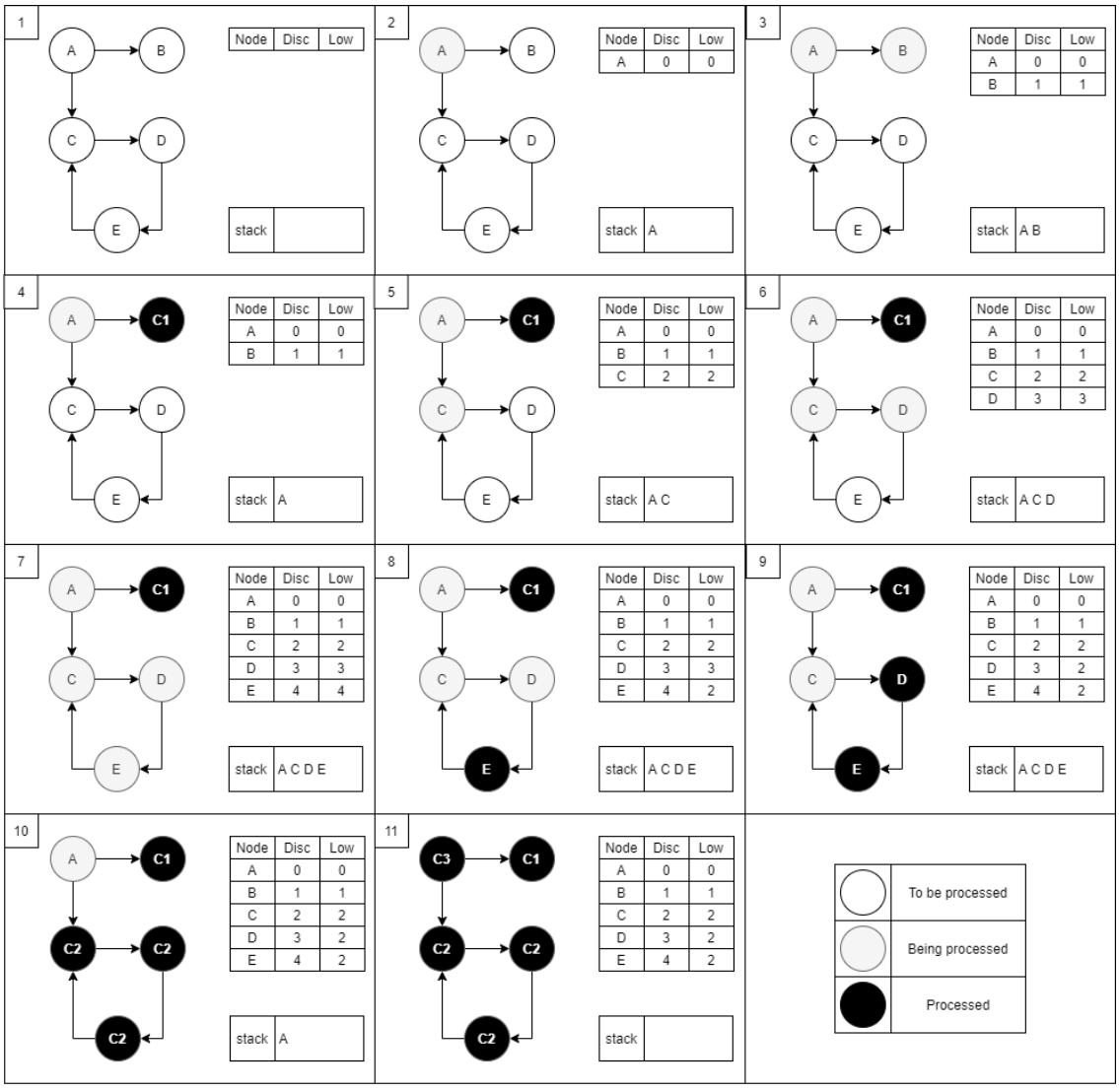


Figure 3.1: Example illustration of Tarjan's algorithm for connected components

# Chapter 4

# Pre-existing proof of the recursive Tarjan's algorithm

The correctness of the recursive version of Tarjan's algorithm for strongly connected components has already been verified in multiple papers. The recursive version of Tarjan's algorithm has been verified using the tools Why3 and Coq in the paper "*Formal proofs of Tarjan's strongly connected components algorithm in Why3, Coq and Isabelle*" by *R. Chen* [4] and *"A semi-automatic proof of strong connectivity"* by *R. Chen* [5]. These articles verify a declarative version of Tarjan's algorithm for strongly connected components. A disadvantage of verifying a declarative version is that the Tarjan's algorithm is often defined and implemented as an imperative algorithm. This means that there is an inherent mismatch between the verified algorithm and the algorithm in practice. *R. Chen* states that the verification of an imperative implementation of Tarjan's algorithm is overly complex which is why the declarative version should be preferred [5]. We believe that this does not have to be the case. Instead of using ghost variables to color the nodes like *R. Chen*, we define our predicates directly in terms of the program variables. This will result in a straightforward and humanly readable proof. This proof will form the foundation of the Dafny verification.

The verification of Tarjan's algorithm has been part of ongoing work at the Eindhoven University of Technology. An imperative implementation of the recursive version of Tarjan's algorithm was already verified in Dafny before the creation of this paper [8]. However there were still problems with this pre-existing verification. The verification turned out to be slow and unstable. Efforts to scale the verification of the recursive algorithm to the iterative algorithm have not been successful up to the creation of this paper. The existing verification of Tarjan's algorithm for SCC did provide a good foundation. In this chapter we first introduce the new verification followed by the discussion of the disadvantages of this pre-existing verification. Lastly we will propose non-disruptive improvements in the Dafny implementation to better suit our needs.

## 4.1 Pre-existing verification of recursive implementation of Tarjan's algorithm

In this section we outline the pre-existing verification of Tarjan's algorithm. The pre-existing proof of the recursive algorithm was initially created as a manual proof. This manual proof was an easy humanly readable proof. Unfortunately, this proof was a monolithic proof which was hard to implement and debug. In order to improve the verification process in Dafny the pre-existing proof had gone trough some iterations. However, the underlying problems with the pre-existing verification have not been resolved up until now. We will like to create a new proof which can easily be verified using Dafny. We want to reuse the clear and humanly readable invariants from the pre-existing proof. Unfortunately the proofs that the invariants are maintained cannot easily

| | |
|---|---|
| $S_0$ | $\forall u \in low : low[u] \leq disc[u]$ |
| $S_1$ | $\forall u \in stack : \exists v \in stack : \mathsf{connected}(u, v) \wedge low[u] = disc[v]$ |
| $S_2(u)$ | $\forall v \in stack : disc[u] < disc[v] \Rightarrow \mathsf{connected}(u, v)$ |
| $S_3$ | $\forall u, v \in stack : u \prec v \Rightarrow disc[u] < disc[v]$ |
| $S_4(u)$ | $\forall v \in stack : disc[u] < disc[v] \Rightarrow low[u] \leq low[v]$ |
| $S_5(u)$ | $u \in stack \iff low[u] \neq disc[u]$ |
| $S_6$ | $\forall u \in low : (u \in stack \oplus \exists C \in result : u \in C)$ |
| $S_7$ | $\forall u \in \mathsf{nodes}(result) : \mathsf{successors}(u) \subseteq \mathsf{nodes}(result)$ |
| $S_8(u)$ | $\forall v \in stack : disc[u] < disc[v] \Rightarrow low[v] \neq disc[v]$ |
| $S_9^{\prec}(u)$ | $\forall v \in stack : disc[u] < disc[v] : \forall w \in \mathsf{successors}(v) : w \in stack \Rightarrow low[v] \leq disc[w]$ |
| $S_9^{\preceq}(u)$ | $\forall v \in stack : disc[u] \leq disc[v] : \forall w \in \mathsf{successors}(v) : w \in stack \Rightarrow low[v] \leq disc[w]$ |
| $S_{10}$ | $\forall C \in result : \mathsf{scc}(C)$ |
| $S_{11}$ | $stack = [\,]$ |
| $S_{12}(u)$ | $stack = \mathsf{old}(stack) \vee stack[\lvert\mathsf{old}(stack)\rvert] = u$ |

Table 4.1: Old invariants recursive Tarjan's algorithm for SSC

be reused. The invariants of the pre-existing verification are shown in table 4.1. Elaboration of each invariant can be found in section 4.1.1. The recursive Tarjan's algorithm annotated with these invariants can be found in Algorithm 4 and 5. Most of these invariants can easily be derived from the program context. The more complex derivations will be explained in 4.1.2.

**Definition 4.1** (Nodes)**.** The function $nodes(result : set\langle set\langle Node\rangle\rangle : set\langle Node\rangle$ combines the set of components to a set of nodes.

$$nodes(result) \equiv \bigcup_{C \in result} C$$

**Definition 4.2** (Precursor operator)**.** The $u \prec v : Node \times Node \to \mathbb{B}$ operator is the precursor operator. This operator returns true if and only if $u$ is located below $v$ in the *stack*.

$$u \prec v \equiv \exists i, j : 0 \leq i < j < |stack| \wedge stack[i] = u \wedge stack[j] = v$$

**Algorithm 4** Tarjan's Strongly Connected Component Algorithm

**Input**: $G = (V, E)$: A graph with nodes $V$ and edges $E$.
**Output**: *result*: A set containing all strongly connected components of the graph.
TARJAN($G$):

```
1:  var stack := [ ]
2:  var low := {:}                          ▷ the empty mapping is denoted as {:}
3:  var disc := {:}                         ▷ u ∈ low means u is a key of mapping low
4:  var result := [ ]
5:  for u ∈ V do
6:      {S0 ∧ S1 ∧ S3 ∧ S6 ∧ S7 ∧ S10 ∧ S11}
7:      if u ∉ low then
8:          StrongConnect(u)
9:          {S5(u) ∧ S12(u)}
10: {∀C ∈ result : scc(C)}                  ▷ soundness
11: {∀u ∈ V : ∃C ∈ result : u ∈ C}          ▷ completeness
12: return result
```

**Algorithm 5** Recursive version of helper function StrongConnect

**Input**: $u$: An element of $V$.
StrongConnect($u$):

1: **var** $k := |disc|$ ▷ $k$ is the discovery time that is assigned to node $u$
2: $disc[u] := k$
3: $low[u] := k$ ▷ initially $low[u] = disc[u]$
4: $stack := stack +\!\!+ [u]$
5: **for** $v \in \mathsf{successors}(u)$ **do**
6: $\{S_0 \wedge S_1 \wedge S_2(u) \wedge S_3 \wedge S_4(u) \wedge S_6 \wedge S_7 \wedge S_8(u) \wedge S_9^{\prec}(u) \wedge S_{10}\}$
7: **if** $v \notin low$ **then**
8: StrongConnect($v$)
9: $low[u] := \mathsf{min}(low[u], low[v])$
10: **else if** $v \in stack$ **then**
11: $low[u] := \mathsf{min}(low[u], disc[v])$
12: **if** $low[u] = disc[u]$ **then** ▷ an SCC has been found
13: $\{S_0 \wedge S_1 \wedge S_2(u) \wedge S_3 \wedge S_4(u) \wedge S_6 \wedge S_7 \wedge S_8(u) \wedge S_9^{\prec}(u) \wedge S_{10}\}$
14: $comp := [\,]$
15: **while** true **do**
16: $v := stack[|stack| - 1]$ ▷ assign the top of the *stack* to $v$
17: $stack := stack[0 : |stack| - 1]$ ▷ pop an element from the stack
18: $comp := comp +\!\!+ [v]$
19: **if** $v = u$ **then break**
20: $result := result +\!\!+ [comp]$
21: $\{S_0 \wedge S_1 \wedge S_3 \wedge S_5(u) \wedge S_6 \wedge S_7 \wedge S_{10} \wedge S_{12}(u)\}$
22: $\{u \in stack \Rightarrow S_2(u) \wedge S_4(u) \wedge S_8(u) \wedge S_9^{\prec}(u)\}$

### 4.1.1 Invariants of the recursive implementation of Tarjan's

The invariants for Tarjan's recursive algorithm can be found in table 4.1. These invariants are the basis of the verification of Tarjan's algorithm. We want to prove that the result of Tarjan's algorithm is sound and complete. A result is sound if every component in *result* is a strongly connected component (invariant $S_{10}$). In the original verification this invariant is proven at line 10 of Algorithm 1. The result is complete if every node is contained in some component of *result*. The completeness proof is performed at line 11 of the main algorithm. In order to prove that the result is sound and complete we require the other invariants. Every invariant follows from an intuitive observation. The idea behind every invariant is shown below:

0. Invariant $S_0$ states that for every node $u$ in *low* it holds that $low[u] \leq disc[u]$. As explained in section 3 $disc[u]$ contains the discovery time of node $u$. The variable $low[u]$ contains the lowest discovery time of a reachable node from $u$. Since every node is connected to itself $low[u] \leq disc[u]$ should always hold.

1. Invariant $S_1$ states that it holds that every node $u \in stack$, $u$ is connected to a node $v \in stack$ such that $disc[v]$ is $low[u]$. This idea reflects the basic idea of the algorithm. For every node $u$ we are searching for a node $v$ with the lowest discovery value. We store the lowest value which has been found in $low[u]$. Note that there are cases where $low[u]$ is never set to the lowest possible value. This can occur if a cycle is found. We can leave $u$ on the stack once we have checked all *successors* of $u$. It can be the case that we later find a new lowest candidate when we are processing another node in the cycle. The value of $low[u]$ is never updated since it will not affect the algorithm.

2. Invariant $S_2$ states that given a node $u$ it holds that for all nodes $v \in stack$, if $disc[u] < disc[v]$ then $u$ and $v$ are connected. This invariant entails that $u$ is connected to all nodes after $u$

on the stack. While this invariant might seem trivial it is actually rather complex. If node $u$ is being processed and we find a new node $v$ then trivially $disc[u] < disc[v]$ and node $u$ is connected to node $v$. However after node $u$ has been processed it does not have to be removed from the *stack*. We also have to prove that node $u$ is connected to all nodes which are added to the *stack* after node $u$ has been processed but before node $u$ is removed from the *stack*.

3. Invariant $S_3$ indicates that if a node $v$ is located after a node $u$ in the *stack* then $disc[u] < disc[v]$ holds. Once a new node is found, we give it a discovery value and we place it on the stack. If node $v$ is discovered after node $u$ then $v$ will be placed after node $u$ in the stack and $v$ will have a higher discovery value.

4. Invariant $S_4$ states that given a node $u$, for any node $v$ with $disc[u] < disc[v]$, it holds that $low[u] \leq low[v]$. This invariant states that a node $u$ has a lower *low* value then all nodes after node $u$ in the stack. This invariant holds when we are actively processing node $u$ in a call `StrongConnect(u)`. Once we push this call onto the recursion *stack*, then this invariant is not maintained until `StrongConnect(u)` is pulled from the recursion *stack*.

5. Invariant $S_5$ states that if a node $u$ is in the stack then $low[u] \neq disc[u]$. This invariant is a postcondition of `StrongConnect(u)`. Node $u$ is removed from the stack once all its successors have been checked but the value of $low[u]$ has not decreased.

6. Invariant $S_6$ states that if a node is in *low* then it is either on the *stack* or in a component of *result*. Note that a node cannot be in a component of *result* and in the *stack* at the same time.

7. Invariant $S_7$ states that for nodes $u \in Nodes(result)$ and all $v \in successors(u)$ it holds that $v$ is also contained in *result*. This invariant holds because we remove a node from the *stack* if and only if all of its successors have been processed and all reachable nodes from $u$ are located after $u$ on the *stack*.

8. Invariant $S_8$ states that for all nodes $v$ after node $u$ in the stack, it holds that $low[v]$ is not equal to $disc[v]$. Invariant $S_8$ is a postcondition of `StrongConnect`. Note that this invariant does not imply invariant $S_5$ since $S_5$ also implies that if $disc[u] \neq low[u]$ then it holds that $u$ is located on the *stack*.

9. Invariant $S_9$ consists of two invariants $S_9^{\preceq}(u)$ and $S_9^{\prec}(u)$.

   The invariant $S_9^{\preceq}(u)$ is another postcondition of `StrongConnect`. Invariant $S_9^{\preceq}(u)$ states that all processed nodes in the *stack* have a *low* value which is lower then the *disc* value of all successors that are also on the *stack*. Note that this invariant holds for all neighbours and not just the neighbours which are still on the *stack*. Unfortunately, this stronger and cleaner invariant is harder to prove. Since we do not need the strong version of $S_9^{\preceq}(u)$ in our proof we use the current version which requires the neighbours to be on the *stack*.

   The invariant $S_9^{\prec}(u)$ is a loop invariant which states that the invariant $S_9^{\preceq}$ holds for all nodes after $u$ in the *stack*.

10. Invariant $S_{10}$ states that all components in *result* are strongly connected components as defined in definition 3.4.

11. Invariant $S_{11}$ states that the *stack* is empty. This invariant is the loop invariant of the main loop of Tarjan's algorithm. This is an important invariant and holds because the *low* value of the first `StrongConnect`$(u_0)$ call can not decrease. Therefore the first call always has to clear the *stack* before terminating.

12. Invariant $S_{12}$ states that the *stack* after a `StrongConnect`$(u)$ call is equal to the *stack* before the call or the first new node on the *stack* is node $u$. This invariant is a postcondition of

`StrongConnect(u)`. This invariant is used to derive two predicates. This invariant ensures that the initial value of the *stack* before the recursive call is maintained. Secondly the invariant ensures that if the *stack* is modified then the first node added to the *stack* is $u$.

### 4.1.2 Monolithic proof of Tarjan's algorithm

In this section the main ideas behind the pre-existing proof are explained. We show how partial correctness can be proven using the annotated algorithm 4 and 5 and the invariants from table 4.1. Partial correctness consists of two parts, soundness and completeness. Soundness guarantees that all components in the result are SCCs. This proof can be further decomposed into a proof for strong connectivity of components, and a proof of maximality. A proof for both of these are given in section 4.1.2.

#### Strong connectivity

A component is a connected component if every node in the component is connected to every other node. At line 20 of the algorithm `StrongConnect`($u$), algorithm 5, a new component $C$ has been found. The component $C$ contains node $u$ and all nodes above node $u$ in the *stack*. We need to establish that this component is strongly connected. To prove that $C$ is a connected component, let $v$ be an arbitrary node in $C$. By invariant $S_1$ we know that $v$ is connected to a node $w$ such that $low[v] = disc[w]$. We know that $low[v] \neq disc[v]$, by invariant $S_8$. Using invariants $S_1$ and $S_3$ we can derive that $w$ is located strictly below $v$ in the *stack*. By invariants $S_4$ and $S_3$ we know that $low[u] \leq low[v]$ and since $low[u] = disc[u]$ we can conclude that $disc[u] \leq disc[w]$. This means that the predicate $connected(v, w)$ holds and we know that $w = u$ or $w$ is between $u$ and $v$ in the stack. This means that $w$ is in $C$. Since $w$ is a node in $C$ we know that $w$ is connected to $u$ or to a node between $u$ and $w$. Since $C$ contains a finite number of nodes we will eventually find a path connecting $v$ to $u$. So we have proven that every node in $C$ is connected to $u$. The other way around, $connected(u, v)$, can easily be derived from invariants $S_2$ and $S_3$. We now conclude that every node in $C$ is connected to $u$ and $u$ is connected to every node in the component. From this we conclude that $C$ is a valid connected component.

#### Maximality

The proof for maximality starts with observations from invariants $S_6$ and $S_9$. These invariants state that every successor of every node in $C$ is located in *low*. Therefore, every successor node is located in the *stack* or in *result*. Let $w$ be a successor of a node $v$ in $C$. By invariants $S_6$ and $S_9$ we know $w \in nodes(result) \cup stack$. We distinguish 2 cases:

- If $w \in nodes(result)$ then it is already part of an $SCC$. By invariant $S_7$ we know that there is no path from $w$ to $v$. So we cannot add $w$ to the component $C$ to create a larger connected component.
- If $w \in stack$ then by invariant $S_9$ we know $low[v] \leq disc[w]$. By invariants $S_4$ and $S_3$ we know that $disc[u] = low[u]$ and we know that $low[u] \leq low[v]$. It follows that $disc[u] \leq disc[w]$. By invariant $S_3$ we can now derive that $w$ is already in $C$, therefore we cannot increase the size of $C$ by including $w$.

Because this covers all cases we can conclude that $C$ is maximal. Since we have shown that $C$ is a connected component in section 4.1.2 we can conclude that $C$ is a valid $SCC$

## 4.2 Pre-existing Dafny implementation

The pre-existing verification from section 4.1 has been implemented in Dafny. This Dafny implementation can be found in appendix A. The problem with implementing this verification is that the verification is a monolithic proof. A Dafny proof is a monolithic proof if the majority of the

verification occurs within the method itself. The problem with a monolithic proof is that the majority of the invariants are in the scope of the Dafny program during the entire proof. Dafny will always try to assert predicates by using the available invariants. Having unneeded invariants will cause an explosion in the number of derivable facts. The explosion of facts can quickly push Dafny to its limits. Once Dafny is pushed to its limits, the verification process will become slow and unpredictable. Minor changes to the implementation can cause the Dafny program to behave unexpectedly. Some seemingly unchanged intermediate facts can stop verifying and the verification time can explode. Furthermore long wait times can greatly hinder the debugging process. In extreme cases Dafny might not even be able to locate the point of failure. Because of these performance problems the proof of termination was omitted from the implementation of the pre-existing verification. The pre-existing implementation consisted of a proof of correctness and a separate proof for termination. Combining these proofs turned out to be infeasible up until now. Scaling the verification from the recursive version Tarjan's algorithm to the iterative version also turned out to be infeasible. This is why we need to resolve any performance issues before we start verifying the iterative version of Tarjan's algorithm.

The pre-existing implementation has not dealt with the performance issues of the Dafny verifier because no techniques were available at that time. The developers pressed on and tried to implement different proofs and different invariants until they found some implementation which verified. This approach is not sustainable and cannot deal with the complexity of the iterative Tarjan's algorithm. In this paper techniques will be introduced which can be used to resolve performance issues for program proofs. If one cannot verify a proof in Dafny then one should structure the proof into smaller proofs. These smaller proofs are called lemmas in Dafny and are defined in definition 4.3.

Lemmas are self-contained proofs which derive a postcondition from some preconditions. Lemmas can be called in a Dafny program. When a lemma is called then Dafny can instantly derive the postconditions from the preconditions without providing a proof. This is usefull when a proof is too complex for the Dafny verifier to verify. We can simplify the complicated large proof into several simple smaller proofs. If we keep splitting the proofs into smaller proofs then we will eventually be able to verify all sub-proofs. These sub-proofs can then be combined into one complete stable proof without performance issues. One should not compromise on clarity and readability to avoid performance issues. Using lemmas we can resolve the performance issues from the verification of the Tarjan's algorithm. Unfortunately it is not easy to identify lemmas which allow us to trivialize the verification of the recursive Tarjan's algorithm. We will dive into this problem in detail in chapter 5. In chapter 6 we will introduce lemmas and other techniques to completely resolve any performance and stability issues. Because the performance issues will be resolved using this technique we no longer need to compromise on the readability and maintainability. In the remainder of this section we are going to introduce changes to improve the usability of the Dafny implementation. Note that the importance of these changes cannot be understated.

An example of how we verify a method using a lemma lets $M$ be a method which returns two integer return values $x, y$ and let $x = y$ be the postcondition of $M$. Now let the derivation of this postcondition be too complex but let us be able to derive that $x \leq y$ and $y \leq x$ hold at the end of the method. We could verify the method using a lemma which proves that $x \leq y$ and $y \leq x$ implies $x = y$. This lemma $L$ would requires two integers $x, y$ as input. We want to prove that if $x \leq y$ and $x \geq y$ then $y = x$ holds. So lemma $L$ has precondition $x \leq y \wedge x \geq y$ and postconditon $x = y$. We can now call $L$at the end of the method $M$. Since Dafny can prove that $x \leq y$ and $y \leq x$ hold, Dafny can prove that the precondition of lemma $L$ holds. Dafny is now able to assume that the postcondition of lemma $L$ holds at the end of method $M$. Because of this Dafny knows that $x = y$ which allows us to verify that the postcondition of method $M$ holds. Note that we also need to verify lemma $L$ by proving that $x \leq y$ and $y \leq x$ implies $x = y$. This is done in a separate smaller proof.

The Dafny code of this example is shown below. In Dafny all preconditions are annotated by the *requires* keyword and all postconditions are annotated by the *ensures* keyword. Note that the statement **assert** $P$ tells Dafny to prove that the predicate $P$ holds.

```
Method M() returns(x: int, y: int)
  ensures x = y
{
  var x, y;
  ...
  assert x ≤ y;
  assert y ≤ x;
  L(x, y);
  return x, y;
}

lemma L(x: int, y: int)
  requires x ≤ y
  requires x ≥ y
  ensures x = y
{
  if x < y {
    assert false;
  } else if x > y {
    assert false;
  } else {
    assert x = y;
  }
}
```

**Definition 4.3** (Lemma)**.** Lemmas are separate Dafny proofs. A lemma has precondition and a postcondition. Next to this a lemma can optionally have input arguments, the precondition and postcondition can depend on these input arguments. A lemma can be verified by providing a proof which derives the postcondition from the precondition. These lemmas can be called in proofs. When a lemma is called then Dafny has to prove that the precondition holds before the lemma call. After the lemma call Dafny can assume that the postcondition holds. By definition lemmas are methods without program code.

### 4.2.1 Refactoring the pre-existing Dafny implementation

As mentioned before, the pre-existing proof will be completely reimplemented. The invariants and ideas behind the proof are reused. Since we are no longer constrained by performance and readability issues we can make changes to the proof to improve the readability of the proof and the implementation process. We will introduce 6 major changes:

1. Introducing types
2. Rewriting invariants
3. Simplifying the termination proof
4. Grouping global variables
5. Introducing modules
6. Creating self-contained lemmas

These changes and their advantages are discussed below.

#### Introducing types

In the original implementation all variables are of the type $\mathbb{N}$ or $seq\langle\mathbb{N}\rangle$. Because of this the developer needs to create many additional predicates which reduce readability. In some places the choice of type can even significantly overcomplicate the proof. One example of a type which greatly improves readability are nodes. A node $u$ was implemented as an element $u : \mathbb{N}$. Every time $u$ is used it should hold that $u <$ NodeLimit, where NodeLimit is the maximum number of nodes. This implementation of *Node* requires the developer to create a predicate for every object

and function which states that a node is below the node limit. This greatly reduces readability and maintainability. A node could also be implemented as:

$$\textbf{type } \mathit{Node} = i : \mathbb{N} \mid i < \textsc{NodeLimit}$$

Note that we have defined *Node* as a **type** and not as a **new type**. This means that Dafny still sees an element of type *Node* as a natural number. This means that a node can be incremented without typecasting. This is useful in the main loop of Tarjan's algorithm.

An example of a type which complicates the proof is the type *Component*. Originally *Component* was defined as a sequence of nodes. This means that result was defined as a sequence of sequences of nodes. $\mathit{result} : \mathit{seq}\langle\mathit{seq}\langle\mathbb{N}\rangle\rangle$. A sequence enforces an ordering. Since components are not ordered we are enforcing an unneeded constraint. Using *set* can greatly simplify proofs so we will be using it where possible. A disadvantage of using a set is that deterministically selecting items from a set is difficult. An explanation of how items can be selected deterministically from a set is shown in appendix B.

$$\textbf{type } \mathit{Component} = \mathit{set}\langle\mathit{Node}\rangle$$

We use the following types for the global variables:

- $\mathit{disc} : \mathit{map}\langle\mathit{Node}, \mathbb{N}\rangle$
- $\mathit{low} : \mathit{map}\langle\mathit{Node}, \mathbb{N}\rangle$
- $\mathit{stack} : \mathit{seq}\langle\mathit{Node}\rangle$
- $\mathit{result} : \mathit{set}\langle\mathit{set}\langle\mathit{Node}\rangle\rangle$

### Rewriting invariants

The invariants in the pre-existing verification have been rewritten to improve the performance of the verification. Since we resolve the performance issues we can select invariants based on usability and readability. We rewrite some of the invariants from table 4.1. We rewrite invariants $S_2$, $S_8$, $S_9$ and $S_{12}$. All updated invariants are shown in table 4.2.

Invariant $S_2(u)$ used to state that node $u$ is connected to all nodes after $u$ in the stack. This invariant always holds for all nodes in the stack and not just for $u$. This is why we have rewritten invariant $S_2$ to state that every node is connected to all nodes after it in the stack. Because of this the input parameter $u$ is no longer required. This stronger invariant is more readable and easier to use but it is harder to prove.

$$S_2 \equiv \forall u, v \in \mathit{stack} : \mathit{disc}[u] \leq \mathit{disc}[v] \Rightarrow \mathsf{connected}(u, v)$$

Invariant $S_8(u)$ is used to derive that for any node $v$ located after $u$ in the stack the predicate $\mathit{low}[v] \neq \mathit{disc}[v]$ holds. This invariant is used to state that for all nodes $v$ in the stack if $\mathit{disc}[u] < \mathit{disc}[v]$ then it holds that $\mathit{low}[v] \neq \mathit{disc}[v]$. By invariant $S_3$, we know that a node in the stack has a *disc* value larger then $u$ if and only if it is located after node $u$ in the stack. Let $u$ be the $i^{th}$ element in the stack such that $\mathit{stack}[i] = u$. We know that all nodes after $u$ in the stack have an index $j$ with $i < j$. Invariant $S_8$ becomes more readable when we use the index of a node instead of the *disc* value. Furthermore this invariant also becomes easier to prove. We use the following definition for invariant $S_8$:

$$S_8(i) \equiv \forall v \in \mathit{stack}[i : |\mathit{stack}| - 1] : \mathit{low}[v] \neq \mathit{disc}[v]$$

Invariant $S_9$ used to consist of 2 invariants. The invariant $S_9^{\prec}(u)$ which stated: Let $v$ be a node in the stack where $\mathit{disc}[u] < \mathit{disc}[v]$ and let $w$ be a node which is in the stack and which is a successor of $v$ then $\mathit{low}[v] \leq \mathit{disc}[w]$. The invariant $S_9^{\preceq}(u)$ implies with any $v$ in the stack where

| | |
|---|---|
| $S_0$ | $\forall u \in low : low[u] \leq disc[u]$ |
| $S_1$ | $\forall u \in stack : \exists v \in stack : \mathsf{connected}(u, v) \wedge low[u] = disc[v]$ |
| $S_2$ | $\forall u, v \in stack : disc[u] \leq disc[v] \Rightarrow \mathsf{connected}(u, v)$ |
| $S_3$ | $\forall u, v \in stack : u \prec v \Rightarrow disc[u] < disc[v]$ |
| $S_4(u)$ | $\forall v \in stack : disc[u] < disc[v] \Rightarrow low[u] \leq low[v]$ |
| $S_5(u)$ | $u \in stack \iff low[u] \neq disc[u]$ |
| $S_6$ | $\forall u \in low : (u \in stack \oplus u \in \mathsf{nodes}(result))$ |
| $S_7$ | $\forall u \in \mathsf{nodes}(result) : \mathsf{successors}(u) \subseteq \mathsf{nodes}(result)$ |
| $S_8(i)$ | $\forall v \in stack[i : \lvert stack\rvert - 1] : low[v] \neq disc[v]$ |
| $S_9(i)$ | $\forall u \in stack[i : \lvert stack\rvert - 1], v \in \mathsf{successors}(u) : v \in low \wedge (v \in stack \Rightarrow low[u] \leq disc[v])$ |
| $S_{10}$ | $\forall C \in result : \mathsf{scc}(C)$ |
| $S_{11}$ | $stack = [\,]$ |
| $S_{12}(u)$ | $stack = \mathsf{old}(stack) \vee \mathsf{old}(stack) +\!\!+ [u] \leq stack$ |

Table 4.2: Invariants recursive Tarjan's algorithm for SSC

$disc[u] \leq disc[v]$. Similar to invariant $S_8$ we can improve the readability of invariant $S_9^{\preceq}(u)$ by giving the index of node $u$ as input. This gives us the following invariant:

$$\forall u \in stack[i : |stack| - 1], v \in \mathsf{successors}(u) : v \in stack \Rightarrow low[u] \leq disc[v]$$

Furthermore, invariant $S_9^{\prec}(u)$ is now equal to $S_9(i + 1)$. We can combine the two complex invariant into one more readable invariant. Lastly we want to further strengthen this invariant by also implying that all neighbours of node $v$ are in *low*. This fact can greatly simplify some proofs. This gives us the final invariant:

$$S_9(i) \equiv \forall u \in stack[i : |stack| - 1], v \in \mathsf{successors}(u) : v \in low \wedge (v \in stack \Rightarrow low[u] \leq disc[v])$$

If we would not strengthen invariant $S_9$ by including $v \in low$ then we would need to prove that $v \in low$ holds before we can apply invariant $S_9$.

Lastly invariant $S_{12}(u)$ used to state that the after a call `StrongConnect(u)` the stack was unchanged or the first new element on the stack was node $u$. The invariant is used to derive that a call from `StrongConnect(u)` does not remove elements from the initial stack. Furthermore, we derive that if elements are added to the stack then the first element which was added to the stack was node $u$. We can derive this invariant using the old invariant $S_{12}$. However we can also change $S_{12}$ to the desired invariant for clarity.

$$S_{12}(u) \equiv stack = \mathsf{old}(stack) \vee \mathsf{old}(stack) +\!\!+ [u] \leq stack$$

Here the old keyword indicates the state of the stack before a call from `StrongConnect`.

### Simplifying the termination proof

When proving correctness of Tarjan's algorithm for strongly connected components we should also prove termination. Because of the performance problems it was assumed that the Tarjan algorithm terminates during the correctness proof. In a separate proof it was then proven that Tarjan's algorithm terminates. Because we are no longer compensating for performance issues we can perform both proofs simultaneously. However, we can also significantly simplify the termination proof. To prove that an algorithm terminates, we should provide some expression which value decreases every iteration of the algorithm. Furthermore, we have to prove that the decreasing expression has a constant lower bound and we need to prove that the decrease has a constant non zero lower bound. In order to prove that the `StrongConnect` function terminates we provide the following termination measure:

$$\textsc{NodeLimit} - |low|$$

Proving that this expression decreases every recursive iteration of `StrongConnect` is trivial. However, we should also prove that this expression has a lower bound. We used to prove this by adding the invariant $|low| <$ NodeLimit. If we can prove that this invariant is maintained then the expression is never negative. However, the variable *low* is a mapping from keys to value. Let the set *Keys* be a set of nodes which contains all keys from *low*. The size of *Keys* is equal to the size of *low*. We can prove that any set of nodes is smaller than NodeLimit. A node is defined as natural numbers smaller than NodeLimit. From this it follows that there are only NodeLimit unique nodes.

Convincing Dafny that any set of nodes has at most NodeLimit elements can be tricky. We can prove this by providing Dafny with a set of nodes $V$ which contains all nodes and has size NodeLimit. Now let $V'$ be an arbitrary set of nodes. Dafny knows that if $|V| < |V'|$ then there exists a node in $V'$ which is not in $V$. However since $V$ contains all nodes we arrive at a contradiction. With this we can conclude that any set of nodes contains at most NodeLimit elements. From that we can conclude that the size of *low* is at most NodeLimit and hence $0 \leq$ NodeLimit $- |low|$.

### Grouping global variables together

Currently we have four global variables in the recursive algorithm: *low*, *disc*, *stack*, *result*. These global variables together make up the state of the algorithm. We introduce a new type *TarjanData* for the state of the algorithm. In the current implementation we also use a type *DiscLow* for the global variables *disc* and *low*. This object helps grouping invariants, however, it is not a vital object. The datatype *DiscLow* is defined in definition 4.4 and the datatype *TarjanData* is defined in definition 4.5

We created this type because we often pass the state of the global variables to different functions and predicates. In our proof we often reason about old states of global variables. Some invariants can hold in a state but aren't maintained when the state is edited. Once these edits are done, the state is updated to restore the invariant. When we want to prove that an invariant is maintained, then we need to reason about all intermediate states and intermediate invariants. Keeping track of the global variables and invariants is a challenge for the developer and for the Dafny verifier. By grouping invariants together we can greatly improve readability and maintainability of the code. Next to this grouping global variables together also decreases the load on the Dafny verifier. Lastly, we use a lot of functions and lemmas in our proof. These functions accept a state as input and return a state as output. If we do not group invariants together then the amount of input fields explodes. These lemmas and functions will be introduced in chapter 6

Grouping the global variables together into one type introduces a change to the algorithm. We argue that this is a minor and acceptable change. If one does not want to make any changes to the algorithm at all then one can also make the state a ghost variable of type *TarjanData*. A ghost variable is a variable which is only used for the verification and cannot influence the behaviour of the algorithm. We could maintain an invariant which states that the global variables are corretly captured by the ghost variable. We have not done this since the gain is minimal and it greatly reduces readability. We do strongly recommend using one variable in the proof to model the entire global state.

**Definition 4.4** (DiscLow)**.** The type *DiscLow* is a data type which contains both the global variables *disc* and *low*. The global state is a data type which means that the variable is an object which can contain fields. The type *DiscLow* variable contains two fields *disc* and *low* which have type $map\langle Node, \mathbb{N}\rangle$.

$$DiscLow \equiv DiscLow(disc : map\langle Node, \mathbb{N}\rangle, low : map\langle Node, \mathbb{N}\rangle)$$

**Definition 4.5** (TarjanData)**.** The type *TarjanData* is a data type which contains all global variables of the Tarjan algorithm. This data type contains three fields. The field *ds* which has type *DiscLow*, the field *stack* of type $seq\langle Node\rangle$ and the field *result* of type $set\langle set\langle Node\rangle$. This

type *DiscLow* is defined in definition 4.4.

$$TarjanData \equiv TarjanData(ds : DiscLow, stack : seq\langle Node\rangle, result : set\langle set\langle Node\rangle\rangle)$$

**Creating self-contained lemmas**

The old implementation omits explicitly passing global variables to predicates and lemmas by directly referring to global variables. This allows the user to omit adding an input field for the global variables. It does not allows the user to omit any preconditions. This is a very minor advantage but it also appeared to make the pre-existing proof more stable. Unfortunately there is a large disadvantage to letting lemmas refer to global variables. Because an input field was omitted the lemmas are no longer self-contained. The lemmas need to be in the scope of the global variables which can hinder reusing or structuring the Dafny code. In the pre-existing implementation the global variables where declared in a class. This required all lemmas to be in a class which directly declared the global variable. Using inheritance the lemmas where separated over different files. This resulted in a large structure of classes which inherited lemmas and variables from each other. These large classes amplified the unpredictable behaviour of the Dafny verifier. In the new verification all lemmas will be self-contained and lemmas will not be part of a class. This prevents unpredictable behaviour and allows us to reusing lemmas.

**Introducing modules**

Once Dafny proofs get larger it becomes useful to divide the code over different files. Dividing the code over different files was a challenge in the pre-existing roof. For performance reasons the pre-existing proof did not pass the global variables to methods but let the lemmas refer to the global variables directly. The global variables were all contained in a class. Because of this all lemmas were also contained in a class. The online way to divide the proof over different files was to declare multiple classes and to use inheritance. As mentioned in the last section we will no longer declare lemmas in classes. By removing the lemmas from classes we can easily separate the proof into different files. We will need to use modules to organize our file system. There are three main advantages of using modules. The first disadvantage is that we can easily find the file which declares a lemma or predicate. The second advantage is that modules prevent name space collisions and the last advantage is that we can easily reuse the lemmas in the iterative proof.

# Chapter 5

# Optimizing complex proofs in Dafny

In Dafny, one can usually verify simple programs using a straightforward approach. However, when the complexity of the verification increases, the verification time explodes and the Dafny verifier becomes unstable. This hinders the debugging and verification process. In order to verify complex programs one should first understand how Dafny verifies a program and why proofs can become unstable.

As discussed in chapter 4, the pre-existing verification of Tarjan's algorithm is unstable and it takes a long time to verify. This occurs because Dafny tries to prove statements using all available invariants. Simply helping Dafny by providing intermediate predicates to guide the Dafny verifier will not resolve these issues. In this section we will introduce techniques which can be used to verify algorithms with unstable proofs. There are two techniques which can be used. These techniques involve creating lemmas and creating opaque predicates. The combination of these two techniques can be used to stabilize unstable proofs. Unfortunately applying these techniques can be complex. In this chapter we explain the techniques and we introduce patterns to efficiently apply these techniques.

## 5.1 The Dafny verifier

The Dafny verifier is the mechanism which Dafny uses to verify Dafny code. In this section we briefly introduce the ideas behind the Dafny verifier. This will allow us to understand why certain proofs can be hard to verify and how to solve them. The Dafny verifier uses the intermediate verification language Boogie. Boogie is a language which can be used to verify simple unstructured code. The Dafny verifier translates Dafny code to Boogie code. Dafny does this by introducing something called dynamic frames. These dynamic frames make the Dafny language more expressive and dynamic frames can be used to hide a large amount of complexity from the developer. The exact inner workings of both Boogie and the Dafny verifier are complex and outside of the scope of this master thesis. In this section we briefly introduce Boogie and dynamic frames.

### 5.1.1 Boogie

Boogie is an intermediate verification language designed to describe verification conditions for static programs. Boogie consists of mathematical and imperative components. The imperative components of Boogie specify sets of execution traces and the desired behaviour.These verification of the execution traces is translated to predicates by Boogie. These predicates are then validated by an SMT solver. More information about using Boogie can be found in the Boogie reference manual [10].

Code is validated by proving that if the precondition holds before the code is executed then the postcondition holds after the code has executed. Boogie and Dafny both verify code based on the theory of Hoare triples as defined by C.A.R.Hoare [7]. A Hoare triple $\{P\}S;\{Q\}$ consist of a predicate $P$, a program $S$ and a predicate $Q$. A Hoare triple states that, given that the predicate $P$ holds before program $S$ then the predicate $Q$ holds after program $S$, given that $S$ terminates. Note that these Hoare triples can be combined. Given that Hoare triples $\{P\}S_1;\{Q\}$ and $\{Q\}S_2;\{R\}$ hold then we can derive that the Hoare triple $\{P\}S_1;S_2;\{R\}$ holds. These Hoare triples are a fundamental building block behind both Dafny and Boogie [13]. Understanding Hoare triples allow Dafny users to tackle complex problems. As an example of Hoare triples take the program **var** $x = y * 3;$. The following Hoare triples are all valid:

$$\{true\}\textbf{var } x := y * 3;\{x = 3 * y\}$$
$$\{0 < y\}\textbf{var } x := y * 3;\{y < x\}$$
$$\{z = 5\}\textbf{var } x := y * 3;\{z = 5\}$$
$$\{true\}\textbf{var } x := y * 3;\{true\}$$
$$\{false\}\textbf{var } x := y * 3;\{false\}$$

Note that some programs might require a precondition in order to have a valid execution trace. Take the program **var** $x := 3/y;$. This program fails if $y = 0$. Because of this the Hoare triple $\{true\}\textbf{var } x := 3/y;\{true\}$ does not hold, but the Hoare triple $\{x \neq 0\}\textbf{var } x := 3/y;\{true\}$ does.

A program $S$ with precondition $Q$ and a postcondition $P$ is valid if the Hoare triple $\{Q\}S;\{P\}$ holds. Boogie proves that $\{Q\}S;\{P\}$ holds by deriving the *weakest precondition*. The weakest precondition is the weakest predicate $Q'$ such that the Hoare triple $\{Q'\}S;\{P\}$ holds. It can now be verified that $S$ confirms to its specification by proving that $Q \Rightarrow Q'$. The proof that $Q \Rightarrow Q'$ can be automated using a SMT solver. Boogie derives the weakest precondition using *weakest precondition calculus* [6, 13].

Directly implementing weakest precondition calculus will result in weakest preconditions which are too complex for a SMT solver. Boogie uses several optimizations to ensure that the weakest precondition can easily be simplified using an SMT solver. Boogie first transforms a program into different blocks of so-called stateless code. Boogie then generates verification criteria for every block of code. If all verification criteria are verified then it follows that the original code is verified. Boogie also builds a table which maps verification criteria to sections of code. This allows Boogie to indicate which section of the code could not be verified. The exact inner workings of Boogie are outside of the scope of this paper. The inner structure of Boogie is explained in the paper *Boogie: A Modular Reusable Verifier for Object-Oriented Programs* [1] and the exact translation of Boogie code to the predicates for the SMT solver is given in the paper *Weakest-Precondition of Unstructured Programs* [2].

### 5.1.2 Weakest precondition calculus

Weakest precondition calculus is a vital building block behind Boogie and therefore Dafny. In this section we will introduce the main ideas behind weakest precondition calculus. For a complete definition of weakest precondition calculus we refer to *The weakest precondition calculus: Recursion and duality* [3] and *Weakest-Precondition of Unstructured Programs* [2]. Let $\mathcal{WP}[S;,\ P]$ be the weakest precondition such that the predicate $P$ holds after the program $S$ assuming that program $S$ terminates. In other words $\mathcal{WP}[S;,P]$ is the weakest predicate such that $\{\mathcal{WP}[S;,\ P]\}\ S;\ \{P\}$ holds. Note that if program $S$ can be split into two programs $S \equiv S_1;S_2;$ then the weakest precondition $\mathcal{WP}[S_1;S_2;,\ P]$ is equal to $\mathcal{WP}[S_1;,\ \mathcal{WP}[S_2;,\ P]]$. This holds because $P$ is the postcondition of $S_2$ so $\mathcal{WP}[S_2;,\ P]$ is the weakest precondition of $S_2$. In order to ensure that this precondition of $S_2$ holds we know that $\mathcal{WP}[S_2;,\ P]$ is the weakest postcondition of $S_1$. From this we conclude that the weakest precondition of $S$ and $S_1$ is $\mathcal{WP}[S_1;,\ \mathcal{WP}[S_2;,\ P]]$. We can also express this using Hoare triples. From the Hoare triples $\{\mathcal{WP}[S_1;S_2;,\ P]\}S_1;\{\mathcal{WP}[S_2;,\ P]\}$ and $\{\mathcal{WP}[S_2;,\ P]\}S_2;\{P\}$ we can derive $\{\mathcal{WP}[S_1;S_2;,\ P]\}S_1;S_2;\{P\}$. As a concrete example of weakest precondition calculus take the program $S$ and postcondition $P$:

$$S \equiv x := 2 * x;\ \textbf{assert } 2 \leq x;\ y := 2/x;$$

$$P \equiv y \leq 1$$

The program $S$ contains an **assert** statement. This statement is an important feature of both Dafny and Boogie. The assert statement **assert** $P'$; states that Dafny should prove that the predicate $P'$ holds. In weakest precondition calculus this statement adds a proof obligation:

$$\mathcal{WP}[\ \textbf{assert}\ P';, Q] \equiv \mathcal{WP}[\emptyset, P' \wedge Q] \equiv P' \wedge Q$$

Here $\emptyset$ is an empty program. The weakest precondition of an empty program is trivially equal to the postcondition.

$$\mathcal{WP}[\emptyset,\ P'] \equiv P'$$

In order to prove that the postcondition $P$ holds after the program $S$ has executed the precondition $\mathcal{WP}[S;\ P]$ has to hold. Substituting $P$ and $S$ with their definitions gives us the weakest precondition:

$$\mathcal{WP}[x := 2 * x;\ \textbf{assert}\ 2 \leq x;\ y := 2/x;,\ y \leq 1]$$

Using weakest precondition calculus we can derive that if $y \leq 1$ has to hold after assignment $y := 2/x$ then the precondition $\mathcal{WP}[y := 2/x;,\ y \leq 1]$ holds before the assignment. A weakest precondition $\mathcal{WP}[y := E;,\ Q]$ can be derived by substituting $y$ in $Q$ with the expression $E$. Next to this the weakest precondition needs to include any preconditions from the expression $E$. This gives us the reduction rule, $\mathcal{WP}[y := E;,\ Q] \equiv \mathcal{WP}[y := E;, true] \wedge Q[y := E]$. Using this reduction rule we can derive that $\mathcal{WP}[y := 2/x;\ y \leq 1]$ is equal to $\mathcal{WP}[y := 2/x;\ true] \wedge (y \leq 1)[y := 2/x]$. This predicate can be simplified to $x \neq 0 \wedge x \leq 2$.

$$\begin{aligned}
&\mathcal{WP}[x := 2 * x;\ \textbf{assert}\ 2 \leq x;\ y := 2/x;\ y \leq 1] \\
\equiv\ &\mathcal{WP}[x := 2 * x;\ \textbf{assert}\ 2 \leq x;,\ \mathcal{WP}[y := 2/x;\ y \leq 1]] \\
\equiv\ &\mathcal{WP}[x := 2 * x;\ \textbf{assert}\ 2 \leq x;,\ \mathcal{WP}[y := 2/x; true] \wedge (y \leq 1)[y := 2/x]] \\
\equiv\ &\mathcal{WP}[x := 2 * x;\ \textbf{assert}\ 2 \leq x;,\ x \neq 0 \wedge 2/x \leq 1] \\
\equiv\ &\mathcal{WP}[x := 2 * x;\ \textbf{assert}\ 2 \leq x;,\ x \leq 2 \wedge x \neq 0]
\end{aligned}$$

The reduction rule for the assert expression has already been introduced. Note that this assert expression enforces a stronger precondition. While this precondition becomes stronger it also allows us to simplify the precondition. It is important to note that weaker preconditions do not result in easier proofs. This is why assertions can greatly speed up proofs. Next to this Dafny and Boogie also return an error when a assert statement fails to verify. This is why assert statements are an important tool to verify programs.

$$\begin{aligned}
&\equiv \mathcal{WP}[x := 2 * x;\ \textbf{assert}\ 2 \leq x;,\ x \leq 2 \wedge x \neq 0] \\
&\equiv \mathcal{WP}[x := 2 * x;,\ \mathcal{WP}[\textbf{assert}\ 2 \leq x;,\ x \leq 2 \wedge x \neq 0]] \\
&\equiv \mathcal{WP}[x := 2 * x;,\ 2 \leq x \wedge x \leq 2 \wedge x \neq 0] \\
&\equiv \mathcal{WP}[x := 2 * x;,\ x = 2]
\end{aligned}$$

The weakest precondition of program $S$ can now be derived by substituting $x$ with the last assignment. This gives us the weakest precondition $x = 1$ which ensures that postcondition $P$ holds after program $S$ has executed, $\{x = 1\}\ x := 2 * x;\ \textbf{assert}\ 2 \leq x;\ y := 2/x;\ \{y \leq 1\}$. So program $S$ is verified if we can derive $x = 1$ from the precondition. Note that the variables $x$ and $y$ are input variables of the program $S$ and not fresh variables. Creating a new fresh variable requires the **var** operator. The reduction rule over the **var** operator introduces existential quantifiers. In order to simplify the example we let the variable $y$ be an input variable of $S$ and the program $S$ sets the value of $y$ to $2/x$;

$$\begin{aligned}
&\equiv \mathcal{WP}[\emptyset, \mathcal{WP}[x := 2 * x;,\ x = 2]] \\
&\equiv (x = 2)[x := 2 * x] \\
&\equiv 2x = 2 \\
&\equiv x = 1
\end{aligned}$$

### 5.1.3 Dynamic frames

Boogie can calculate a weakest precondition for simple programs. However, Boogie cannot calculate the weakest preconditions of complex programs. Boogie cannot derive the weakest precondition over method calls or over most loops. Next to this Boogie also does not support complex types. Dafny resolves these problems by adding something called *dynamic frames* [9]. Dynamic frames allow Dafny to use partial proofs. Dafny allows us to provide a specification for a type, loop or method. When code is verified which contains a loop or a method call then Dafny assumes that the specification of the loop or method holds. Dafny can then use the specification to derive a weakest precondition. Unfortunately this process is quite complex and outside of the scope of this paper. The paper *Specification and Verification of Object-Oriented Software* gives a good practical introduction of dynamic frames and the translation of Dafny to Boogie [11]. The paper *Dafny: An Automatic Program Verifier for Functional Correctness* introduces the theory behind a lot of Dafny features depending on dynamic frames [12].

As an example of dynamic frames let $M$ be a verified method with precondition $Q$ and postcondition $P$ and let $S$ be the program $x := 0;\ M();$. Now trivially the Hoare triple $\{Q\}M;\{P\}$ holds. Let's say we want to prove that $x \leq 1$ holds after program $S$. The postcondition $P$ of method $M$ will likely contain information about the effect of method $M$ however $P$ does not have to contain any information about the variable $x$. Therefore the Hoare triple $\{Q\}M;\{P\}$ is too weak to create a weakest precondition for most postconditions of the program $S$.

This problem can be resolved using dynamic frames. A dynamic frame contains the *footprint* of a class, method or loop. The *footprint* contains all variables which are used by the class method or loop. This dynamic frame can then be used to prove that all predicates which depend on variables which are not in the footprint are maintained. Take the postcondition $x \leq 1$ of method $S$ and let the footprint of method $M$ be the new variable $z$. We can now prove that when a predicate $R$ which does not depend on variable $z$ holds before a method call $M$ then the predicate $R$ holds after the method call $M$. We can use dynamic frames and weakest precondition calculus to derive the weakest precondition $\mathcal{WP}[x := 0;\ M();, x \leq 1]$.

In order to calculate the weakest precondition we need to create a reduction rule for method $M$. If no reduction rule exists for method $M$, we replace $M$ with an alternative expression for which we have a reduction rule. In order to create this expression we require two new statements. The boogie statement **assume** $P$ and the new variable statement **var**. The assume statement states that the predicate $P$ holds but the statement does not add a new proof obligation. If a predicate $R$ should hold after this statement then the weakest precondition of **assume** $P$ is $P \Rightarrow R$. This gives us the reduction rule:

$$\mathcal{WP}[\textbf{assume } P;,\ R] \equiv \mathcal{WP}[\emptyset,\ P \Rightarrow R]$$

The second statement **var** $z$ introduces a new variable $z$. This statement gets translated to boogies **havoc** statement. The **havoc** statement states that $z$ has an arbitrary value. This statement can be combined with the **assume** statement to derive the weakest preconditions more efficiently. Since this statement is rather complex we will instead use a simple theoretical definition of **var**. The expression **var** $z$ creates a new variable $z$. We know that if a predicate $R$ holds after an expression **var** $z$ then for all values $z'$ of variable $z$ the predicate $R[z := z']$ has to hold. This gives us the reduction rule for creating a new variable.

$$\mathcal{WP}[\textbf{var } z;,\ R] \equiv \mathcal{WP}[\emptyset, \forall z' : R[z := z']]$$

Using the assumption expression and the new variable expression we can create a reduction rule replacing the method call $M();$. We replace the method call $M();$ by the statement
**assert** $Q$; **var** $z$; **assume** $P$;. This statement adds the new proof obligation the precondition $Q$ of method $M$, introduces a new variable $z$ for the footprint of $M$ and assumes that the postcondition $P$ holds.

$$\mathcal{WP}[M();,\ T] \equiv \mathcal{WP}[\textbf{assert } Q;\ \textbf{var } z;\ \textbf{assume } P;,\ T]$$

Using these new reduction rules we can calculate the weakest precondition of $S$. Note that because $x$ is not in the footprint of $M$ we know that the value of $x$ does not affect the predicates $Q$ and $P$ so $Q[x := 0]$ is simply $Q$ and $P[z := z'][x := 0]$ is simply $P[z := z']$.

$$\mathcal{WP}[x := 0;\ M();,\ x \leq 1]$$

$$\equiv \mathcal{WP}[x := 0;\ \textbf{assert}\ Q;\ \textbf{var}\ z;\ \textbf{assume}\ P;,\ x \leq 1]$$

$$\equiv \mathcal{WP}[x := 0;\ \textbf{assert}\ Q;\ \textbf{var}\ z;,\ \mathcal{WP}[\textbf{assume}\ P;,\ x \leq 1]]$$

$$\equiv \mathcal{WP}[x := 0;\ \textbf{assert}\ Q;\ \textbf{var}\ z;,\ P \Rightarrow x \leq 1]$$

$$\equiv \mathcal{WP}[x := 0;\ \textbf{assert}\ Q;,\ \mathcal{WP}[\textbf{var}\ z;\ P \Rightarrow x \leq 1]]$$

$$\equiv \mathcal{WP}[x := 0;\ \textbf{assert}\ Q;,\ \forall z' : (P \Rightarrow x \leq 1)[z := z']]$$

$$\equiv \mathcal{WP}[x := 0;\ \textbf{assert}\ Q;,\ \forall z' : P[z := z'] \Rightarrow x \leq 1]$$

$$\equiv \mathcal{WP}[x := 0;,\ \mathcal{WP}[\textbf{assert}\ Q;,\ \forall z' : P[z := z'] \Rightarrow x \leq 1]]$$

$$\equiv \mathcal{WP}[x := 0;,\ Q \wedge \forall z' : P[z := z'] \Rightarrow x \leq 1]$$

$$\equiv \mathcal{WP}[\emptyset,\ (Q \wedge \forall z' : P[z := z'] \Rightarrow x \leq 1)[x := 0]]$$

$$\equiv Q \wedge \forall z' : P[z := z'] \Rightarrow 0 \leq 1$$

$$\equiv Q \wedge true$$

$$\equiv Q$$

These dynamic frames can also be used to calculate the weakest precondition over a loop. Calculating the weakest precondition over a loop is similar to calculating the weakest precondition over a method call. Dafny requires the user to provide a specification for the loop. The loop can then be replaced with an assert and an assume statement based on the specification. The user needs to provide a separate proof which proves that the specification of the loop holds. The Dafny user provided a specification of a method by defining a precondition and a postcondition. The user does not need to define a precondition and a postcondition for the specification of a loop. The user is only required to define a *loop invariant*. Using this loop invariant Dafny can derive the precondition and the postcondition of the loop.

A loop invariant $I$ is an invariant which is maintained by every iteration of a loop. Given a loop **while** $B$ **do** $S_1$;**od** and a loop invariant $I$ the loop invariant is valid if the Hoare triple $\{B \wedge I\}\ S_1;\{I\}$ holds. Once we have proved that $\{I\}\ S_1;\{I\}$ holds then it follows that following the Hoare triple holds:

$$\{I\}\ \textbf{while}\ B\ \textbf{do}\ S_1;\ \textbf{od}\ \{I \wedge \neg B\}$$

Intuitively this holds because when an invariant $I$ holds before the loop and every iteration of the loop maintains $I$ then invariant $I$ holds once the loop has terminated. Note that since the **while** loop executes as long as the predicate $B$ holds it follows that $\neg B$ holds once the loop terminates. Remember that Hoare triples assume that the a program terminates. Dafny requires a separate proof which proves that the loop terminates. Once we have proven that the loop invariant $I$ holds then we can replace the loop with an assert and an assume statement similar to a method call. As an example take the program:

$$S_0;\ \textbf{while}\ B\ \textbf{do}\ S_1;\textbf{od};\ S_2;$$

When we provide a loop invariant $I$ then we are required to provide a separate proof which proves that $\{I\}\ S_1;\{I\}$ holds. It then follows that $\{I\}$ **while** $B$ **do** $S_1$; **od** $\{I \wedge \neq B\}$ holds. Using this Hoare triple we can simplify the program by replacing the loop by an assert statement of the precondition $I$ and an assume statement of the postcondition $I \wedge \neg B$. This gives us the alternative form of the program without the **while** loop:

$$S_0;\ \textbf{assert}\ I;\ \textbf{assume}\ I \wedge \neg B;\ S_2;$$

We have now introduced a rough overview of the Dafny verifier. Once proofs get complex then the weakest precondition will get too large. Once the weakest precondition gets too large then the verifier becomes slow and the verifier can fail to verify proofs. Using assert statements, the weakest precondition can be simplified and the proof can be debugged. Dafny will always inform us which assert statements can be verified and which statements cannot be verified. So we can detect which predicates cannot be verified and which facts are verified. Unfortunately just using assert statements is insufficient to verify very complex proofs. At some point the weakest precondition can become very large. By adding assert statements to the code we can strengthen the weakest precondition. Strengthening the weakest precondition might simplify the derivation of the weakest precondition from the precondition. Unfortunately it is not always possible to create a stable proof by just adding assert statements. Furthermore changes to the code might cause statements which use to verify to stop verifying unexpectedly. Once this happens we say that the verify has become unstable. One should not try to continue debugging unstable proofs. It is often quicker to use techniques to stabilize the proof then to press on. In the remainder of this chapter we will introduce techniques to stabilize proofs and we will apply them to the recursive version of Tarjan's algorithm.

## 5.2 Adding lemmas

When the verification of a proof is unstable and slow then we can improve the verification process by adding lemmas. As defined in definition 4.3, lemmas are a Dafny feature which allows us to split a proof into smaller proofs. Lemmas allow us to split a proof into smaller proofs. There are several advantages to this. The most important advantage is that we can significantly decrease the complexity of the weakest precondition. Next to this lemmas have several other advantages. The main advantages are improving the readability of the proof, facilitating recursion and facilitating code reuse. A lemma consists of a set of input variables, a sequence of preconditions, a sequence of postconditions and a proof. The proof should derive the postconditions from the preconditions. A lemma can then be called in other proofs. When a lemma is called then Dafny can use the fact that the precondition implies the postcondition without deriving the postcondition. Dafny does this by replacing the call of a lemma with precondition $Q$ and postcondition $P$ with the statement **assert** $Q$; **assume** $P$;. This is similar to a method call because a lemma is actually just a method without program code. Therefore, a lemma has an empty footprint and a lemma does not have any runtime behaviour.

Large proofs can be verified by adding assert statements to prove intermediate invariants. These intermediate invariants help Dafny to derive a stronger weakest precondition. Strengthening the weakest precondition assists the SMT solver in deriving the weakest precondition from the precondition. Unfortunately, this derivation can still be too complex. When a large proof is unstable then we can replace a complicated derivation of an intermediate invariant with a lemma. This lemma should prove that the intermediate invariant holds based on a precondition. Dafny now no longer needs to derive that the intermediate invariant holds in the program proof, Dafny does need to prove that the precondition of the lemma holds. This lemma can greatly improve the stability of the large proof. Next to this Dafny does need to verify the lemma in a separate proof. Dafny needs to derive the intermediate invariant from the precondition of the lemma. The precondition only needs to contain information which is required to derive the intermediate invariant which limits the load on the SAT solver. Because of this deriving the intermediate invariant in a lemma is a lot more stable then deriving it in the program proof. Overall adding lemmas greatly improves the stability of the sub-proof and somewhat improves the stability of the large top-level proof.

As an example of how a lemma is used let's say that we want to prove that the complicated predicate $P$ holds after a program $S$. Let us assume that Dafny can derive $P$ from an intermediate predicate $P'$ and let Dafny be able to derive $P'$ from a simple predicate $P''$ which is simple enough such that Dafny can prove that $P''$ holds after $S$. We can provide a Dafny proof which proves that predicate $P$ holds after $S$ by adding the assert statemens $S$;**assert** $P''$;**assert** $P'$;**assert** $P$;

to $S$. The weakest precondition from this proof is:

$$\mathcal{WP}[S; \textbf{assert } P''; \textbf{assert } P'; \textbf{assert } P;,\ P] \equiv \mathcal{WP}[S;,\ P'' \wedge P' \wedge P]$$

The SAT solver might be able to derive that predicate $P$ holds after $S$ since we guide the proof somewhat using intermediate invariants. Unfortunately, if the derivation is complex enough then the SAT solver will still struggle with the complex weakest precondition. We can create a lemma $L$ to simplify the large precondition. Lemma $L$ should have precondition $P''$ and postcondition $P$. The derivation of predicate $P$ can be replaced by lemma $L$. We can now verify that $P$ holds after $S$ by appending a method call $L()$ after $S$.

$$\mathcal{WP}[S; L();\ P] \equiv \mathcal{WP}[S; \textbf{assert } P'';\ \textbf{assume } P;\ P] \equiv \mathcal{WP}[S; P'' \wedge (P \Rightarrow P)]$$

This new weakest precondition can easily be proved by the SAT solver since $P \Rightarrow P$ is simply *true*. Next to this the intermediate predicate $P'$ no longer needs to be derived in the program proof. We still need to prove that lemma $L$ is correct by proving that $P''$ implies $P$. We can prove that this lemma is correct by proving that $P''$ implies $P'$ and $P'$ implies $P$.

This general approach often works however we can encounter two problems. The first problem is that just using lemmas might not be sufficient to create a stable top-level proof. This can occur if the weakest precondition is still very large and requires a lot of invariants which cannot be derived from other simpler invariants. This problem can be resolved by adding opaque predicates which will be explained in section 5.3. The second problem is that it can be hard to identify potential lemmas. A lemma just proves that a precondition implies a postcondition. It is often easy to split a derivation in a lemma into several cases or smaller derivations. Unfortunately it can be hard to identify lemmas when we are proving that a method is correct. In section 5.2.1 a pattern is introduced which creates lemmas to stabilize the verification of a method. The lemmas introduced in section 5.2.1 also allow us to create lemmas which stabilize the program proof even more then normal approach from this section.

### 5.2.1 Introducing lemmas to methods

Lemmas are usually used in the following straightforward pattern. When we want to verify a Hoare triple $\{Q\}S;\{P\}$ then we can define a lemma $L$ with precondition $R$ and postcondition $P$. Using lemma $L$ we can verify the Hoare triple $\{Q\}S;\{P\}$ by proving that $Q \Rightarrow \mathcal{WP}[S; L;,\ P]$ holds. Using weakest precondition calculus we get

$$\mathcal{WP}[S; L;,\ P] \equiv \mathcal{WP}[S; \textbf{assert } R; \textbf{assume } P;,\ P] \equiv \mathcal{WP}[S;,\ R]$$

So essentially using lemma $L$ we can prove that the Hoare triple $\{Q\}S;\{P\}$ holds if Hoare triple Hoare triple $\{Q\}S;\{R\}$ holds since lemma $L$ proves that $R$ implies $P$. Unfortunately it is not always possible to find a predicate $R$ such that the proof of $\{Q\}S;\{R\}$ is easy. In order to verify Tarjan's algorithm we will require stronger lemmas. In this section we will introduce a pattern to define lemmas which does not require us to define an intermediate predicate $R$.

We want to define lemmas which immediately proves that a Hoare triple holds for a section of code. Given a Hoare triple $\{Q\}S;\{P\}$ we want to define a lemma which states that if $Q$ holds before $S$ then $P$ holds after $S$. If $S$ contains only one operations then this is straightforward. Let $S \equiv x := x + 1$, we can now define a lemma $L(x')$ with precondition $Q[x := x']$ and postcondition $P[x := x' + 1]$. We can verify $\{Q\}x := x + 1;\{P\}$ using weakest precondition calculus and lemma $L$:

$$\mathcal{WP}[L(x); x := x + 1;,\ P] \equiv \mathcal{WP}[L(x);\ P[x := x + 1]]$$

$$\equiv \mathcal{WP}[\textbf{assert } Q[x := x];\ \textbf{assume } P[x := x + 1],\ P[x := x + 1]]$$

$$\equiv \mathcal{WP}[\textbf{assert } Q;,\ \mathit{true}] \equiv Q$$

Essentially we have moved the proof of the program $S$ completely into a lemma. Because of this the proof of $S$ has become very stable. If the program $S$ was part of a larger program then the

proof of the larger program has also become a lot more stable. Once the proof is contained in a lemma then we can usually easily split the proof into smaller proofs and resolve any performance issues. We will briefly discuss these techniques later for now it is enough to define lemmas which move the complexity from the program proof to a lemmas.

The pattern of substituting the value of $x$ with the effect of program $S$ works well if $S$ is very small. Unfortunately, once $S$ becomes larger it becomes unfeasible to directly substitute $x$ with the effect of all operations. Next to this defining one lemma for every operation is also not feasible. We create lemmas which reason about the effect of multiple consecutive operations by introducing functions. The function should consist of the same operations as the program $S$. This function can then be used to define a lemma which reasons about the effect of the program. Let $S$ be a program, let $V$ be the set of all input variables of $S$ and let $M$ be the set of all variables which are changed or created by $S$. Now let $f$ be a function which accepts the set of values $V'$ of all variables $V$ as input and which returns a set of values $M'$ of all variables $M$ as output. It should hold that for every value of variables $V$ before $S$ the return value of $f(V')$ is equal to the values of $M$ after $S$. Using this function we can define lemmas which prove that $\{Q\}S;\{P\}$ holds even if $S$ is large. More concretely let's say we want to prove that the Hoare triple $\{Q(v_1,\dots v_i)\}S;\{P(m_1,\dots m_j)\}$ holds where $v_1,\dots v_i$ are input variables of $S$ and $m_1,\dots m_j$ are the new and modified variables. We can define a lemma $L(v_1,\dots v_i)$ with precondition $Q(v_1,\dots v_i)$ and with postcondition $P(f(v_1,\dots v_i))$. This lemma can now be used in a method to derive $\{Q(v_1,\dots v_i)\}S;\{P(m_1,\dots m_j)\}$.

As an example let $S$ be a program $S \equiv x := x+1;$ **var** $z := y*x;$ with precondition $Q(x,y)$ and postcondition $P(x,y,z)$. We want to verify that $\{Q(x,y)\}\ x := x+1;$ **var** $z := y*x;\{P(x,y,z)\}$ holds using a lemma. In order to define this lemma we introduce a function:

$$f(x,y) \equiv \textbf{var } x' := x;\ x' := x'+1;\ \textbf{var } z := y*x';\ \textbf{return } (x',z');$$

Note that functions cannot modify input variables so we need to create fresh variables $x', z'$. Using function $f$ we can define lemma $L(x,y)$ with precondition $Q(x,y)$ and postcondition $P(f(x,y).x', y, f(x,y).z')$. The program $S$ can now be verified using lemma $L$. The proof that the effect of lemma $f$ is equal to program $S$ is usually trivial for Dafny. Using weakest precondition calculus the weakest precondition becomes:

$$\mathcal{WP}[L(x,y);\ x := x+1;\ \textbf{var } z := y*x;,\ P(x,y,z)]$$

$$\equiv \mathcal{WP}[L(x,y);,\ P(x+1,y,y*(x+1))]$$

$$\equiv \mathcal{WP}[\textbf{assert } Q(x,y);\ \textbf{assume } P(f(x,y).x', y, f(x,y).z');,\ P(x+1,y,y*(x+1))]$$

$$\equiv \mathcal{WP}[\textbf{assert } Q(x,y);,\ P(f(x,y).x', y, f(x,y).z') \Rightarrow P(x+1,y,y*(x+1))]$$

By definition of $f$ it follows that $f(x,y).x = x+1$ and $f(x,y).y = y*(x+1)$ so $P(f(x,y).x', y, f(x,y).z') = P(x+1,y,y*(x+1))$ and the SAT solver should be able to derive that $P(f(x,y).x', y, f(x,y).z') \Rightarrow P(x+1,y,y*(x+1))$ is *true*. The proof that the effect of $f$ is equal to the effect of $S$ usually easily within Dafny's capabilities.

$$\equiv \mathcal{WP}[\textbf{assert } Q(x,y);,\ true]$$

$$\equiv Q(x,y)$$

### Defining functions

We have now introduced a pattern to create a lemma which proves that a Hoare triple holds. In order to prove that a Hoare triple $\{Q(x,y)\}S;\{P(x,y,z)\}$ holds we need to define a function $f$ with the same effect as program $S$. One might wonder how we can define such a function. Unlike programs, functions cannot call methods, contain loops and modify input variables. Next to this functions contain a different *if* statement then programs. When a program contains a loop or a method call then we generally cannot create a function with the same effect. If a program contains no loops, method calls or if statements then we can use a simple pattern to create a function. Let

$S$ be a program without loops, method calls or if statements. If program $S$ does not modify input variables then we can directly use $S$ to define $f$. We can define function $f$ as $S$; **return** $(x, y, z)$;. This simple definition does not work when $S$ modifies an input variable. We can circumvent this problem by creating new local variable in $f$ and using the local variable instead of the input variables. Lets say that function $S$ modifies variable $x$. We can define $f$ by first creating a new variable $x'$ with initial value $x$. Since $x'$ is not an input variable of $f$ we can change variable $x'$. We can now define a new program $S'$ which is identical to program $S$ but where variable $x$ is replaced by variable $x'$. We can now define $f$ as **var** $x' := x$; $S'$; **return**$(x', y, z)$. We can always use this approach to create functions for programs without loops, method calls or if statements. In the example of the previous section program $S$ was defined as $x := x + 1$; **var** $z := y * x$;. We can create function $f$ by introducing the new variable $x'$. We can now define $f$ as:

$$f(x, y) \equiv \textbf{var } x' := x;\ x' := x' + 1;\ \textbf{var } z := y * x';\ \textbf{return } (x', y, z);$$

Note that variable $y$ is not modified so we could also chose to not return it. Next to this we know that function $f$ is deterministic since program $S$ needs to be deterministic. Because of this we can declare $f$ as a *function method*. A function method is a Dafny feature which allows us to declare a function as both a method and a function. Essentially a function method is an executable function which can be called from methods and from code annotations. In practice we can simplify the verification and debugging processes by replacing $S$ by a call to function method $f$. Calling the function is slightly less performance intensive and improves readability. Once the code verifies we can replace the function call again with the original code $S$.

As mentioned before we cannot define a function if $S$ contains a loop or a method call. We cannot create a lemma to reason about the effect of $S$. The best alternative is to verify $S$ using multiple lemmas. We can create one lemma for every section of code without a loop or method call. We can verify the entire program by combining these lemmas. As an example, when we want to prove that the Hoare triple $\{Q\}\ S_1;\ \textbf{while } B \textbf{ do } S_2; \textbf{od } S_3;\ \{P\}$ holds then we cannot create one function for the entire program. We can create a stable proof by creating lemmas for every section of code $S_1, S_2, S_3$. In our example the Dafny user needs to create 3 lemmas and a loop invariant $I$. The first lemma should prove that the Hoare triple $\{Q\}S_1;\{I\}$ holds. The second lemma should prove that $\{I \wedge B\}S_2;\{I\}$ holds and the third lemma which should prove that $\{I \wedge \neg B\}S_2;\{P\}$ holds. These 3 lemmas allow us to move as much complexity to lemmas as possible. Using these 3 lemmas Dafny should be able to derive that the entire program is correct.

The last case is that $S$ contains an *if* statement. In a function every branch of the if statement needs to return a value. We can technically create a function even if $S$ contains an *if* statement. Unfortunately, a function containing *if* statements can be tedious to verify. In practice it is often easier to split $S$ into smaller programs which do not contain *if* statements.

### Difference lemma pattern and straight forward method verification

When we want to verify a program $S$ with precondition $Q$ and postcondition $P$ then the first step is always defining a specification for all loops and method calls within $S$. We can then divide program $S$ into multiple subprograms $S_1, \ldots, S_n$ without method calls and loops. For every subprogram $S_i$ we can specify a Hoare triple $\{Q_i\}S_i;\{P_i\}$. These Hoare triples and the specifications can then be combined to verify program $S$. For every program $S_i$ in $S$ we would need to prove:

$$\textbf{assert } Q_i;\ S_i;\ \textbf{assert } P_i;$$

If the proof that $P_i$ holds is complex then we need to assist Dafny. The classic approach would be to derive intermediate invariants within $S_i$. We can add these intermediate invariants to $S_i$ by adding assert statements to $S_i$. We can also assist Dafny by defining simple lemmas which prove that a simple intermediate invariant implies a complex intermediate invariant. These lemmas can then be used to simplify the derivation of $P_i$.

An alternative approach is defining a lemma using the pattern from this chapter. We can define a function with the same effect as $S_i$. Using this function we can now define a lemma which proves that if $Q_i$ holds before $S_i$ then $P_i$ holds after $S_i$. This would give us the Dafny code:

$$\textbf{assert } Q_i;\ L();\ S_i;\ \textbf{assert } P_i;$$

As discussed before this second aproach using lemma $L$ creates a more stable proof then just adding lemmas and assert statements to $S_i$.

#### Advantages and disadvantages of the lemma pattern

In the previous section we have introduced a pattern which moves the verification of a program to a lemma. The main advantage of this approach is that that the verification of the program proof does not depend on the definition of $P$ and $Q$. Almost all complexity is moved to the lemma. Lemmas are easier able to verify because it is usually easy to split a lemmas into different lemmas. The easiest and most common approach is splitting the postcondition of a lemma. If we want to verify a lemma which states that $Q$ implies $P$ then we can define two predicates $P_1$ and $P_2$ such that $P_1 \wedge P_2 \Rightarrow P$. We can define two smaller lemmas, one with postcondition $P_1$ and one with postcondition $P_2$. These smaller lemmas might not require all information from $Q$ so we could also try to weaken the preconditions $Q_1, Q_2$ by removing unneeded information. Simplifying the precondition reduces the load on the SAT solver which greatly improves performance. More concretely when a lemma $L$ which proves that $Q \Rightarrow P$ is unstable then we can split the proof into two smaller lemmas $L_1, L_2$. Lemma $L_{1,2}$ proves that $Q_{1,2}$ implies $P_{1,2}$, where preconditions $Q_1, Q_2$ are defined such that $Q \Rightarrow Q_1 \wedge Q_2$ holds and the postcondition are defined such that $P_1 \wedge P_2 \Rightarrow P$ holds. Using lemmas $L_1, L_2$ we can then create a stable proof of the larger lemma $L$. There are multiple other approaches which might work depending on the exact proof however the best approach differs on a case to case basis.

There is however one remaining problem, the pattern of defining lemmas which verify a Hoare triples might not be sufficient to create a stable program proof. When we use a lemma which verifies that a Hoare triple $\{Q\}S;\{P\}$ then the program proof does not depend on the definition of $Q$ and $P$. Unfortunately, Dafny does not know this. Dafny will hand the weakest precondition to the SMT solver. The SMT can verify that the program is correct by proving that the function and the program have the same effect. However the SMT solver will also try to prove that the program is correct by exploring other options. The SMT solver will substitute the pre and postcondition with their exact definitions. If these definitions are complex then the SMT solver will still be unable to finish the proof. Because of this the proof can still be unable. In order to ensure that the verification of the method is stable even if the pre and postcondition are complex we need to introduce opaque predicates. Opaque predicates will be defined in section 5.3

## 5.3 Opaque predicates

In section 5.2.1, a strategy was introduced to stabilize the verification of a method by introducing lemmas. This strategy might not be sufficient to create a stable proof. We can further stabilize the proofs by introducing opaque predicates. An opaque predicate is a predicate which exact definition is hidden from Dafny. The meaning of an opaque predicate can be revealed using the *reveal* keyword. Revealing a predicate allows us to manage where Dafny has access to definition of the predicate. Declaring a predicate as opaque can in turn results in a huge performance increase. Dafny proves program correct using an SMT solver. The SMT accept a predicate as input which states that the precondition implies the weakest precondition. The solver will try to prove that the predicate is equal to *true*. If the input predicate is huge then the SMT solver will not be able to derive *true*. This input predicate can contain other predicates. The SMT solver can replace these predicates by their definition. Sometimes this is not required and this increases the load of the SMT solver unnecessarily. By hiding the definition of predicates we can prevent the SMT solver from wasting time working with definitions which are not required. Note that Dafny can call a lemma with an opaque predicate as precondition without revealing the predicate.

Unfortunately, declaring predicates as opaque forces the Dafny user needs to manually reveal predicates. This requires a lot of time and effort from the developer. Furthermore, revealing a

predicate results in a small performance penalty. These problems can be mitigated by grouping predicates together in invariant groups. An invariant group $P$ containing predicates $P_1, \dots P_n$ is in and of itself an opaque predicate which definition is a conjunction of all predicates in the invariant group. So the invariant group $P$ is an opaque predicate with definition $\wedge_{i\in[1,n]} P_i$. The advantage of an invariant groups is that all predicates in the group can be revealed in one go. Selecting invariant groups is a vital step in the verification process. A clever selection of the invariant groups can significantly reduce the complexity of the proof and improve the readability of the verification. Note that we can add invariant groups into other invariant groups. This becomes useful when invariant groups become very large.

Opaque predicates are a powerful tool especially used together with lemmas. Let $S$ be a program which modifies some variable $v$. We want to prove that the Hoare triple $\{Q_1(v) \wedge \dots \wedge Q_m(v)\}S;\{P_1(v) \wedge \dots \wedge P_n(v)\}$ holds. We can prove that this Hoare triple holds by introducing a lemma using the pattern from section 5.2.1. We define a function $f(v)$ with the same effect as program $S$ and a lemma $L(v)$ with precondition $Q_1(v) \wedge \dots \wedge Q_m(v)$ and postcondition $P_1(f(v)) \wedge \dots \wedge P_n(f(v))$. Using lemma $L$ Dafny can prove that the Hoare triple holds by proving that $f$ and $S$ have the same effect. A problem can occur when the pre and postcondition get too complex. The SMT solver used by Dafny will get bogged down trying to work with the definition of the pre and postcondition even though it is not required. We can try to declare every predicate $Q_1, \dots P_n$ as opaque. This would resolve the performance issues however this would add a lot of extra work for the Developer. The developer would need to reveal every predicate individually and this would quickly become unfeasible in practice. To avoid this issue we define two large invariant groups $Q \equiv Q_1 \wedge \dots \wedge Q_m$ and $P \equiv P_1 \wedge \dots \wedge P_n$. This would create a stable program proof and the developer can reveal the definition of $P$ and $Q$ immediately when verifying the lemma $L$. It is still advisable to verify lemma $L$ by using smaller lemmas. We could verify lemma $L$ by creating one lemma for every postcondition $P_i$.

It might not be immediately clear why the definition of the predicates are not required and why it can bog down the SAT solver that much. It is important to remember that Dafny calculates a weakest precondition and then uses a SAT solver to prove the weakest precondition can be derived from the precondition. If this derivation is complex then trivially the SAT solver will take a long time. We can create a weakest precondition which is easier to derive from the precondition by adding lemmas. However when the precondition and weakest precondition are very large then the SAT solver can have trouble even if the derivation might seems simple to a human. Take the Hoare triple $\{Q(v)\}S;\{P(v)\}$ with lemma $L$ from the previous example and let $v'$ be the value of variable $v$ after $S$. Using lemma $L$ the weakest precondition would be:

$$\mathcal{WP}[L(v);\ S;,\ P(v)]$$

$$\mathcal{WP}[\textbf{assert}\ Q(v);\ \textbf{assume}\ P(f(v));,\ P(v)[v := v']]$$

$$\mathcal{WP}[\textbf{assert}\ Q(v);,\ P(f(v)) \Rightarrow P(v)[v := v']]$$

$$Q(v) \wedge (P(f(v)) \Rightarrow P(v)[v := v'])$$

So the Hoare triple holds if the predicate $Q(v) \Rightarrow (Q(v) \wedge (P(f(v)) \Rightarrow P(v)[v := v']))$ is equal to *true*. By hand this proof is trivial, the predicate $P(v)[v := v']$ is $P(v')$ and $P(v')$ is equal to $P(f(v))$. From this we conclude that $P(f(v)) \Rightarrow P(v)[v := v'])$ is equal to *true* which allows us to simplify the predicate to $Q(v) \Rightarrow Q(v) \wedge true$ which is trivially equal to *true*. The SAT solver can use the same proof, so the SAT solver has to derive that the effect of $S$ is equal to $f$ and then the SAT solver can simplify the entire predicate to *true*. If $P$ and $Q$ are not opaque then the SAT solver can attempt to unfold these definitions. Unfolding $P$ and $Q$ would turn our initial expression $Q(v) \Rightarrow (Q(v) \wedge (P(f(v)) \Rightarrow P(v)[v := v']))$ into:

$$(Q_1(v) \wedge \dots \wedge Q_m(v)) \Rightarrow$$

$$(Q_1(v) \wedge \dots \wedge Q_m(v) \wedge ((P_1(f(v)) \wedge \dots \wedge P_n(f(v))) \Rightarrow (P_1(v) \wedge \dots \wedge P_n(v))[v := v'])$$

Substituting $P$ and $Q$ by its exact definitions only complicates the task for the SAT solver. The SAT solver can now also try to substitute the value of $v$ in all sub-expressions of $P$ and $Q$. We know that all these simplifications do not bring the SAT solver closer to its goal however the SAT solver does not know this. The SAT solver can explore a huge state space even if it is not necessary. By declaring $P$ and $Q$ as opaque the SAT solver cannot unfold the predicates and the SAT solver will quickly find the correct derivation.

The addition of the opaque predicate $P$ and $Q$ makes the top level verification a lot more stable. Creating a stable proof of lemma $L$ is easier then creating a stable program proof of $S$. We create one lemma $L_i(v)$ with postcondition $P_i(f(v))$ and the required invariants from $Q(v)$ as precondition. These techniques are sufficient to verify both versions of Tarjan's algorithm. It is important to note that defining these lemmas and opaque predicates is time consuming. These techniques can be used in a wide range of proofs however they should only be used when needed for performance reasons. Adding opaque predicates is especially time consuming. I would advice only using opaque predicates if just adding lemmas and functions is insufficient to verify a program.

# Chapter 6

# Stable proof of the recursive Tarjan's algorithm

In chapter 5 techniques are introduced which allow us to create stable proofs. In this chapter we apply these techniques to verify the recursive Tarjan's algorithm. In order to stabilize the proof we need to identify functions and invariant groups. These functions and invariant groups can be used to define lemmas. We would like to reuse the lemmas from the verification of the recursive Tarjan's algorithm in the iterative version. The lemmas can be reused because there are a lot of similarities between the recursive and iterative version of Tarjan's algorithm. Because the iterative algorithm is a lot more complex we will make sure the the top-level verification of the recursive algorithm is as stable as possible. The more stable the recursive verification is the easier, the verification of the iterative algorithm will be.

## 6.1 Defining functions in Tarjan's algorithm

We want to define lemmas to verify the `StrongConnect` method of Tarjan's algorithm. We want these lemmas to follow the pattern introduced in chapter 5 where one lemma verifies a Hoare triple by introducing a function with the same effect as the code of the Hoare triple. We can create multiple of these lemmas to verify the recursive `StrongConnect` helper method, algorithm 2. In order to define these lemmas we first need to define functions. These functions have the same effect as a section of code in the `StrongConnect` method. There are four sections of code where the global variables are modified. These sections also occur in the iterative version of Tarjan's algorithm, algorithm 3. We want to create lemmas which reason about the effect of these sections of code. In order to create these lemmas we introduce four functions. These functions return an element of type *TarjanData* as output. The data type *TarjanData* was defined in definition 4.5. This datatype contains four fields: *stack*, *disc*, *low*, *result*, each field corresponds to one global variable. We introduce the following four functions:

1. $AddNewNode(data : TarjanData, u : Node) : TarjanData$: The *AddNewNode* function is defined in algorithm 6. The effect of this function corresponds to the effect of lines 1 to 4 in the recursive `StrongConnect` method. This section of code accepts the current state *data* and a node $u$ as input and returns a new state containing $u$ as output. This output state is the same as the input state but with a new node $u$ added to the global variables. Note that technically this section of code also creates the local variable $k$ as output. However since this variable is never used after this section we do not need to let the *AddNewNode* function return it.

2. $UpdateFromLow(data : TarjanData, u : Node, v : Node) : TarjanData$: The *UpdateFromLow* function is defined in algorithm 7 and the effect corresponds to the effect of line 8 in the

recursive `StrongConnect` algorithm. This function accepts a state *data* and two nodes $u, v$ as input. This function updates the state such that the value $low[u]$ is the minimum of its current value and the value $low[v]$.

3. *UpdateFromDisc*($data : TarjanData, u : Node, v : Node$) : *TarjanData*: The *UpdateFromDisc* function is defined in algorithm 8. The effect of this function corresponds to the effect of line 10 in the `StrongConnect` helper function. This function accepts a state *data* and two nodes $u, v$ as input. This function then updates the value $low[u]$ to the minimum of itself and $disc[v]$.

4. *PopFromStack*($data : TarjanData, i : \mathbb{N}$) : *TarjanData*: The function *PopFromStack* is defined in algorithm 9. The function has the same effect as the second while loop in the the `StrongConnect` method, lines 11 to 18. This function accepts a state *data* and a natural number $i$ as input, the function then removes nodes from the stack until the $i^{th}$ node has been removed from the stack. The function creates a new component with all removed nodes. This definition of the *PopFromStack* function is not the same as the code from line 11 to 18. In the algorithm we remove nodes from the stack until node $u$ is removed. We can prove that every node on the stack is unique and we know the index of node $u$ on the stack. So we can prove that *PopFromStack* function has the same effect as the code from line 11 to 18. We use this definition of *PopFromStack* since this definition is significantly less cumbersome to work with then a recursive function or a loop invariant. Furthermore, note that the variable *stack* is a sequence of nodes and a component is a set of nodes, therefore we need to cast the new component from $seq\langle Node\rangle$ to $set\langle Node\rangle$. This cast function is defined in algorithm 15 from appendix B.

**Algorithm 6** The AddNewNode function

**Input**: *data*: The global state of type *TarjanData*.
**Input**: $u$: An element of *Node*.
AddNodeStart($data, u$):

```
1: var k := |data.disc|
2: data.disc[u] := k
3: data.low[u] := k
4: data.stack := stack ++ [u]
5: return data
```

**Algorithm 7** The UpdateFromLow function

**Input**: $u, v$: Elements of *Node*.
**Input**: *data*: The global state of type *TarjanData*.
UpdateFromLow($data, u, v$):

```
1: data.low[u] := min(data.low[u], data.low[v])
2: return data
```

**Algorithm 8** The UpdateFromDisc function

**Input**: $u, v$: Elements of *Node*.
**Input**: *data*: The global state of type *TarjanData*.
UpdateFromDisc($data, u, v$):

```
1: data.low[u] := min(data.low[u], data.disc[v])
2: return data
```

**Algorithm 9** The PopFromStack function

**Input**: $i$: Amount of nodes to be removed from the stack $0 < i < |data.stack|$.
**Input**: *data*: The global state of type *TarjanData*.
PopFromStack($data, i$):
1: $C :=$ SeqToSet($data.stack[i : |data.stack|]$)
2: $data.result := data.result \cup C$
3: $data.stack := data.stack[0 : i]$
4: **return** *data*

## 6.2 Introducing predicates groups and lemmas

In the previous section several functions were introduced. These function allow us to create lemmas which reason about the effects of sections of Tarjan's algorithm. We can now create lemmas which reason about the effect of the update sections. Unfortunately these functions are not sufficient to create a stable proof. As discussed in chapter 5 we can optimize proofs further by introducing invariant groups. We distinguish four invariant groups in the recursive version of Tarjan's algorithm. These groups are:

1. *TarjanData.Valid*: This group contains all invariants which should hold for any state of the global variables.

2. *LoopInvariant*: This group of invariants contains the loop invariants from the first loop of the recursive `StrongConnect` helper function, line 5 to 10 in algorithm 2. Once this loop has completed we should be able to derive the postcondition of `StrongConnect` using this invariant group.

3. *PostconditionStrongConnect*: This group of invariants contains all invariants which should hold at the end of a call of `StrongConnect`. These invariants are also used to prove that a recursive call maintains the loop invariants from the first loop in the `StrongConnect` function.

4. *PreconditionStrongConnect*: This group of invariants contains all invariants which should hold before a call `StrongConnect`.

The invariant groups contain predicates from table 4.2 and some additional context information. The context information consists of small invariants which are required to prove that an invariant from table 4.2 is maintained. In this section we will also briefly introduce the lemmas which are required to prove that the invariant groups are maintained. The exact definitions of these lemmas are given in appendix C.

### 6.2.1 TarjanData Valid invariants group

This *TarjanData.Valid* invariant group contains invariants which hold in every global state. All invariants in this group should be maintained when the global variables are edited. The *TarjanData.Valid* invariant group is defined in definition 6.1. Because the size of this invariant group is rather large, we have created two sub invariant groups *DiscLow.Valid* and *Result.Valid*. These invariant groups are part of the invariant group *TarjanData.Valid* and they have only been added to improve the readability within lemmas. The invariant group *DiscLow.Valid* contains all invariants which only reason about the global variables *disc* and *low*. Similarly the invariant group *Result.Valid* contains invariants which only reason about the global variable *result*.

In every global state the invariant *TarjanData.Valid* should hold. The global variables are updated in the `StrongConnect` method. These updates occur in the four sections of the `StrongConnect` method. We need to prove that every section maintains the invariant *TarjanData.Valid*. We create one lemma for each section. This gives us the following 4 different lemmas:

1. *AddNewNodeDataValid*(*data* : *TarjanData*, *u* : *Node*) with postcondition *AddNewNode*(*data*, *u*).*Valid*

2. *UpdateFromLowDataValid*(*data* : *TarjanData*, *u* : *Node*, *v* : *Node*) with postcondition *UpdateFromLow*(*data*, *u*, *v*).*Valid*

3. *UpdateFromDiscDataValid*(*data* : *TarjanData*, *u* : *Node*, *v* : *Node*) with postcondition *UpdateFromDisc*(*data*, *u*).*Valid*

4. *PopFromStackDataValid*(*data* : *TarjanData*, *i* : $\mathbb{N}$) with postcondition *PopFromStackDataValid*(*data*, *i*).*Valid*

These lemmas require the same input variables as their respective functions. Every lemma proves that its respective function maintains all invariants in *data.Valid*

**Definition 6.1** (TarjanDataValid)**.** The *TarjanDataValid* invariant group contains invariants from table 4.2. The invariant group *TarjanData.Valid* contains invariants which should hold in any state. *TarjanData.Valid* contains the following invariants:

- *DiscLow.Valid*(): This invariant is an invariant group which states that the global variables *disc* and *low* are in a valid state. This invariant group is defined in definition 6.2.

- *Result.Valid*: This invariant is an invariant group which states that the variable *result* is in a valid state. This invariant group is defined in definition 6.3.

- $S_1$: This invariant states that, for every node $u \in$ *stack* there exists a connected node $v \in$ *stack* such that *low*[$u$] = *disc*[$v$] holds.

- $S_3$: This invariant states that for every pair of nodes $u, v \in$ *stack* if $v$ is located after node $u$ in the stack then *disc*[$u$] < *disc*[$v$] holds.

- $S_6$: This invariant states that every node $u \in$ *low* it holds that $u$ is either contained in *result* or in *stack*

**Definition 6.2** (DiscLowValid)**.** The invariant group *DiscLow.Valid* contains invariants which always hold for the *DiscLow* variable. This group contains the following invariants:

- $S_0$: This invariant states that for every node $u \in$ *low* it holds that *low*[$u$] is smaller or equal to *disc*[$u$].

- *disc.Keys* = *low.Keys*: This invariant states that the same nodes are contained in both *disc* and *low*. This predicate provides some additional required context information which is not part of any invariant.

- $\forall u \in disc : disc[u] < |disc|$: This predicate provides some additional context information which is needed to prove that the invariants $S_3$ is maintained.

**Definition 6.3** (ResultValid)**.** The invariant group *Result.Valid* contains invariants which always hold for the *result* variable. This group contains the following invariants:

- $S_7$: This invariant states that all successors of every node in *result* are contained in a component of *result*.

- $S_{10}$: This predicate states that all components of *result* are strongly connected components as defined in definition 3.4.

### 6.2.2 Loop invariants group

This invariant group *LoopInvariant*(*oldData* : *TarjanData*, $u$ : *Node*, *data* : *TarjanData*, $i$ : $\mathbb{N}$) contains all loop invariants from the first loop in the `StrongConnect` helper function, line 5 to 10 of algorithm 2. The invariants in *LoopInvariant* depend on 5 input variables. The input variable *oldData* which is the state at the start of a call `StrongConnect(u)`, $u$ is the node which is being processed, *data* is the current state and $i$ is the number of nodes which have been processed. This loop invariant is used to derive the postcondition of `StrongConnect`

The proof that the *LoopInvariant* is maintained is very complex. One lemma is required to prove that the lemma holds at the start of a recursive call. Beside this we require three lemmas to prove that the invariant is maintained in every iteration of the loop. There are three paths which can be taken in every loop iteration. One lemma is required for each path. This gives us the following four lemmas:

1. *AddNewNodeLoopInvariant*(*data* : *TarjanData*, $u$ : *Node*). This lemma ensures that the loop invariant holds at the start of the loop. The section before the loop invariant is the *AddNewNode* section. This lemma uses the definition of *AddNewNode* and the precondition of `StrongConnect` to ensure that the loop invariant holds.

2. *RecursionLoopInvariant*(*oldData* : *TarjanData*, *recStartData* : *TarjanData*, *data* : *TarjanData*, $u$ : *Node*, $v$ : *Node*, $i$ : $\mathbb{N}$): This lemma ensures that the loop invariant is maintained in the case that node $v$ is not contained in *low*. Here node $v$ is the $i{+}1^{th}$ successor of node $u$. When node $v$ is not in *low* then node $v$ needs to be processed before we continue processing node $u$. A recursive call `StrongConnect`($v$) is performed to process node $v$. A lot of invariants are not maintained by the recursive call. These invariants are restored by a *UpdateFromLow* section. This creates a large complex proof. This proof depends on 3 states. The state *oldData* which is the state at the start of the `StrongConnect` call, *recStartData* which is the state at the start of the recursive call and the state *data* which is the data after the recursive call.

3. *UpdateFromDiscLoopInvariant*(*oldData* : *TarjanData*, *data* : *TarjanData*, $u$ : *Node*, $v$ : *Node*, $i$ : $\mathbb{N}$): This lemma ensure that the loop invariant is maintained if node $v$, the $i + 1^{th}$ successor of $u$, is not contained in *data.low* but node $v$ is contained in *data.stack*. In this case the *UpdateFromDisc* section is executed to update the global variables. This lemma proves that the update *UpdateFromDisc*(*data*, $u$, $v$) maintains the loop invariant.

4. *MaintainsLoopInvariant*(*oldDatadata* : *TarjanData*, *data* : *TarjanData*, $u$ : *Node*, $v$ : *Node*, $i$ : $\mathbb{N}$): This lemma ensures that the loop invariant is maintained in the remaining case where $v$ is not contained in the *stack* and $v$ is not contained in *low*. In this case the global state is not updated. This lemma proves that the loop invariant is maintained without updating the global variables.

**Definition 6.4** (LoopInvariant)**.** The invariant group *LoopInvariant*(*oldData* : *TarjanData*, $u$ : *Node*, *data* : *TarjanData*, $i$ : $\mathbb{N}$) contains the loop invariants from the first loop in the recursive `StrongConnect` method, line 5 to 10 in algorithm 2. This invariant group contains the following invariants:

- *oldData.Valid*: This invariant states that the initial state of the algorithm is in a valid state.
- *data.Valid*: This invariant states that the algorithm is in a valid state. This invariant is an invariant group which is defined in definition 6.1.
- *DataMaintained*(*oldData*, *data*): This invariant states that the values in global variables *disc* and *low* are maintained by the loop. The predicate *DataMaintained* is defined in definition 6.5.
- $S_2$(*data*): This invariant states that every node on the stack is connected to all nodes after it on the stack.

- $S_4(data, u)$: This invariant states that all nodes in the stack after $u$ have a *low* value which is smaller or equal to $low[u]$.

- $S_8(data, |oldData.stack| + 1)$: This invariant states that all processed nodes in the *stack* have a *low* value which is not equal to the *disc* value.

- $S_9(data, |oldData.stack| + 1)$: This invariant states that all processed nodes in the stack have a *low* value which is lower then the *disc* value of all its successors.

- $i \leq |successors(u)|$: This invariant states that $i$ value is not too large.

- $oldData.stack + [u] \leq data.stack$: This invariant provides some context information which states that $u$ is the first node added to the stack at the start of the recursive call.

- $|oldData.stack| + 1 < |data.stack| \Rightarrow data.stack[|oldData.stack| + 1] \in successors(u)$: This invariant states if there is a node directly after $u$ in the stack then the node is a successor of node $u$. From other lemmas we can derive that $data.stack[|oldData.stack|]$ is $u$.

- $\forall v : v \notin oldData.stack \wedge v \in oldData.low \Rightarrow v \notin data.stack$: This invariant contains some context information. This invariant states that no nodes are added back to the stack once they have once been removed.

**Definition 6.5** (DataMaintained)**.** If the predicate *DataMaintained*(*oldData* : *TarjanData*, *newData* : *TarjanData*) holds then the values of *oldData.disc* and *oldData.low* are also contained in *newData.disc* and *newData.low*.

- $\forall u \in oldData.low : u \in newData.low \wedge oldData.low[u] = newData.low[u]$

- $\forall u \in oldData.disc : u \in newData.disc \wedge oldData.disc[u] = newData.disc[u]$

### 6.2.3 Postcondition invariant group

The invariant group *PostconditionStrongConnect*(*oldData* : *TarjanData*, *newData* : *TarjanData*, $u$ : *Node*) contains the invariants which should hold after a call of the `StrongConnect` method. These invariants depend on the node $u$ which is the node that is being processed and the global states *oldData* and *newData* which are the states at the start and at the end of the call. The invariant group *PostconditionStrongConnect* is defined in definition 6.6.

The invariant group *PostconditionStrongConnect* needs to be derived from the invariant group *LoopInvariant*. The *LoopInvariant* has been designed to make this derivation as easy as possible. We require two lemmas to derive *PostconditionStrongConnect* from *LoopInvariant*. At the end of the `StrongConnect` method all nodes can be left on the stack or the top of the stack can be removed and added as a new component. A call `StrongConnect`($u$) creates a new component if $disc[u] = low[u]$ and the algorithm leaves all nodes on the *stack* if $disc[u] \neq low[u]$. We require one lemma for each of the two cases. These lemmas are:

- *Proof1PostconditionStrongConnect*(*oldData* : *TarjanData*, *data* : *TarjanData*, $u$ : *Node*): This lemma handles the case where $data.disc[u] = data.low[u]$ and a new component is made. Here *oldData* should be the state before the recursive call, *data* should be state after the first loop and $u$ should be the node which is being processed. This lemma then proves that the postcondition holds once we remove node $u$ from the stack and add it to a new component.

- *Proof2PostconditionStrongConnect*(*oldData* : *TarjanData*, *data* : *TarjanData*, $u$ : *Node*): This lemma handles the case where $data.disc[u] \neq data.low[u]$. Here *oldData* should be the start state, *data* the current state and $u$ the node which is being processed. Since the state will no longer be updated we prove that the postcondition already holds for the current state

**Definition 6.6** (PostconditionStrongConnect)**.** The invariant group *PostconditionStrongConnect*( *oldData* : *TarjanData*, *newData* : *TarjanData*, $u$ : *Node*) contains all invariants which hold at the end of a recursive call `StrongConnect(u)`. The invariants in this invariant group depend on the node $u$ and the initial and final state of the call `StrongConnect(u)`. The invariant *PostconditionStrongConnect* contains the following invariants:

- *DataMaintained*(*oldData*, *newData*): This invariant states that the values in *oldData.disc* and *oldData.low* are maintained by the call of `StrongConnect(u)`. This predicate is defined in definition 6.5
- $S_2$(*newData*): This invariant states that every node on the stack is connected to all nodes after it on the stack.
- $u \in newData.stack \Rightarrow S_4(newData, u)$: This invariant states that if $u$ is on the *stack* then all nodes in the *stack* after $u$ have a *low* value which is smaller or equal to *low*[$u$]. Note that if $u$ is not in the *stack* then the *stack* is unchanged from the initial stack.
- $S_5$(*newData*, $u$): This invariant states that if and only if node $u$ is still in the *stack* after the call then *low*[$u$] is not equal to *disc*[$u$].
- $S_8(newData, |oldData.stack|)$: This invariant states that for all new nodes in the *stack* it holds that the *low* value is not equal to the *disc* value.
- $S_9(newData, |oldData.stack|)$: This invariant states that all new nodes in the stack have a *low* value which is lower then the *disc* value of all their successors.
- $S_{12}(oldData, u, newData)$: This invariant states that the stack of *oldData* is also fully contained in the stack of *newData*. Furthermore, this invariant states that if *newData.stack* is not equal to *oldData.stack* then $u$ is the first new node on the stack.
- $u \in newData.low$: This invariant provides some missing context information. This invariant simply states that node $u$ is added to *low*.
- $\forall v : v \in oldData.stack \wedge v \in oldData.low \Rightarrow v \notin newData.stack$: This invariant states that no nodes which have been processed are added back to the *stack*.

### 6.2.4 Precondition invariant groups

The invariant group *PreconditionStrongConnect* contains all invariants which should hold before a call of `StrongConnect`. This invariant is defined in definition 6.7. This invariant group is small and does not have to be implemented as an opaque predicate. We have chosen to define this invariant group to improve the readability of the proof. We require only one lemma to prove that *PreconditionStrongConnect* holds before a recursive call. This lemma is *PreconditionStrong -ConnectHolds*(*data* : *TarjanData*, $u$ : *Node*, $v$ : *Node*). Here *data* is the current state, node $u$ is the node which is being processed and node $v$ is the successor of $u$. Node $v$ is also the input node of the recursive call. This covers all invariant groups and lemmas which are required to verify the recursive `StrongConnect` method

**Definition 6.7** (PreconditionStrongConnect)**.** The invariant group *PreconditionStrongConnect*( *data* : *TarjanData*, $u$ : *Node*) contains the preconditions for a call of `StrongConnected(u)`.

1. *data.Valid*: This is the predicate states that the start state is valid. This predicate is defined in definition 6.1
2. $u \notin data.low$: Trivially `StrongConnected(u)` should not be called if node $u$ is already being processed or has already been processed.
3. $S_2(AddNewNode(data, u))$: This predicate looks complex but states that $u$ is connected to all nodes in the stack. Proving that this predicate holds for every recursive call can be complex.

## 6.3 Complete verification

In section 6.2 lemmas and invariant groups were introduced, these lemmas and invariant groups can be used to implement a stable verification of the recursive version of the `StrongConnect` method. This allows us to verify the recursive version of Tarjan's algorithm in Dafny. Tarjan's algorithm is valid if the end state is valid and all nodes are contained in the end state. We prove this using the loop invariant *MainLoopInvariant* from definition 6.8. This invariant is the loop invariant of the main method of Tarjan's algorithm and it states that the state is valid and all processed nodes are contained in the state. We have now defined all lemmas and invariants which are required to verify the recursive version of Tarjan's algorithm. The recursive Tarjan's algorithm annotated with invariant groups and lemmas is shown in algorithm 10 and 11. In the annotated algorithm of `StrongConnect` the state **old**(*data*) is the state at the start of the method and **recStart**(*data*) is the state at the start of a recursive call.

The new proof of the recursive Tarjan's algorithm does not look similar to the pre-existing monolithic proof. A side effect of using lemmas is that the new proof is a lot more abstract. This is very useful in the verification of the iterative algorithm. As we will discuss in chapter 7, the entire verification of the recursive algorithm can be reused in the verification of the iterative algorithm. One might wonder how the pre-exisitng proof relates to the new proof and how the lemmas in the new proof are verified. All lemmas which are used in the `StrongConnect` method prove that an invariant group is maintained. These lemmas use one sub-lemma for each non trivial invariant in the invariant group. Every sub-lemma contains a stable proof for an invariant in table 4.2. These proofs are usually straightforward and will not be discussed in detail. In section 6.3.1 the proof of one invariant from table 4.2 is described in great detail. All other invariants follow the same pattern.

**Definition 6.8** (MainLoopInvariant)**.** The group of invariants *MainLoopInvariant*(*data* : *TarjanData*, $u$ : *Node*) contains the loop invariants from Tarjan's algorithm for Strongly Connected Components, algorithm 1.

1. $data.Valid()$: This is the invariant group that holds in a valid state of Tarjan's algorithm for SCC. The invariant group is defined in definition 6.1

2. $S_{11}(data)$ This predicate states that the stack is empty.

3. $\forall v \in [0..u) : v \in data.ds.low$: This predicate states that all processed nodes from the main loop are contained in low.

**Algorithm 10** Tarjan's Strongly Connected Component Algorithm with proof constructs

**Input**: $G = (V, E)$: A graph with nodes $V$ and edges $E$.
**Output**: *result*: A set containing all strongly connected components of the graph.
**postcondition** *result.Valid*()
TARJAN($G$):
1: **var** *data* := *TarjanData*(*DiscLow*(*map*[ ], *map*[ ], *Result*({ }), *stack*([ ]))
2: **for** $u \in V$ **do**
3: **invariant** *MainLoopInvariant*(*data*, $u$)
4: **if** $u \notin low$ **then**
5: STRONGCONNECT($u$)
6: **return** *data.result*

**Algorithm 11** Recursive version of helper function StrongConnect with proof constructs

**Input**: $u$: An element of *Node*.

**precondition** *PreconditionStrongConnect*$(data, u)$

**postcondition** *PostconditionStrongConnect*(**old**$(data)$, $data, u)$

StrongConnect$(u)$:

1: **lemma** *AddNewNodeDataValid*$(data, u)$ ▷ **old**$(data)$
2: **lemma** *AddNewNodeLoopInvariant*$(data, u)$
3: **var** $k := |data.disc|$ ▷ Section *AddNewNode*$(data, u)$
4: $data.disc[u] := k$
5: $data.low[u] := k$
6: $data.stack := data.stack +\!\!+ [u]$
7: **for** $v\_index \in [0..|\mathsf{successors}(u)|)$ **do**
8: **invariant** *LoopInvariant*(**old**$(data)$, $data, u, v\_index$)
9: $v := \mathsf{successors}(u)[v\_index]$
10: **if** $v \notin data.low$ **then**
11: **lemma** *PreconditionStrongConnectHolds*$(data, u, v)$ ▷ **recStart**$(data)$
12: StrongConnect$(v)$
13: **lemma** *UpdateFromLowDataValid*$(data, u, v)$
14: **lemma** *RecursionLoopInvariant*(**old**$(data)$, **recStart**$(data)$, $data, u, v, v\_index$)
15: $data.low[u] := \mathsf{min}(data.low[u], data.low[v])$ ▷ Section *UpdateFromLow*$(data, u, v)$
16: **else if** $v \in data.stack$ **then**
17: **lemma** *UpdateFromDiscDataValid*$(data, u, v)$
18: **lemma** *UpdateFromDiscLoopInvariant*(**old**$(data)$, $data, u, v, v\_index$)
19: $data.low[u] := \mathsf{min}(data.low[u], data.disc[v])$ ▷ Section *UpdateFromDisc*$(data, u, v)$
20: **else**
21: **lemma** *MaintainsLoopInvariant*(**old**$(data)$, $data, u, v, v\_index$)
22: **if** $data.low[u] = data.disc[u]$ **then**
23: **lemma** *PopFromStackDataValid*$(data, |$**old**$(data).stack|)$
24: **lemma** *Proof1PostconditionStrongConnect*(**old**$(data)$, $data, u)$
25: $comp := [\,]$ ▷ Section *PopFromStack*$(data, |$**old**$(data).stack|)$
26: **while** true **do**
27: $v := data.stack[|data.stack| - 1]$
28: $data.stack := data.stack[0 : |data.stack| - 1]$
29: $comp := comp +\!\!+ [v]$
30: **if** $v = u$ **then break**
31: $data.result := data.result +\!\!+ \{comp\}$
32: **else**
33: **lemma** *Proof2PostconditionStrongConnect*(**old**$(data)$, $data, u)$

### 6.3.1 Proving invariant S1

In this chapter invariant groups and lemmas are defined which can be used to create a stable verification of the recursive Tarjan's algorithm. Each lemma contains a proof that all invariants in an invariant group are maintained. In turn the lemmas use one sub-lemma per invariant to create a stable proof. In this section the verification of one invariant from an invariant group is shown as an example. The verification of all other invariants follow the same pattern. In this section we show how invariant $S_1$, from invariant group *data.Valid*, is verified in Tarjan's algorithm, algorithm 10 and 11 Invariant $S_1(data : TarjanData)$ is defined as :

$$S_1(data) \equiv \forall u \in data.stack : \exists v \in data.stack : \mathsf{connected}(u, v) \wedge data.low[u] = data.disc[v]$$

This invariant $S_1$ states that for every node $u$ in the *stack*, there exists a node $v$ in the *stack* such that $u$ is connected to $v$ and $low[u] = disc[v]$ holds. Invariant $S_1$ is part of the invariant group *data.Valid*. At the start of the main algorithm, algorithm 10, the *stack* is initialized to the empty *stack*. Invariant $S_1$ trivially holds if the stack is empty. After the *stack* is initialized invariant $S_1$ is maintained in the loop. The loop invariant *MainLoopInvariant* implies that *data.Valid* and therefore $S_1$ holds. From the postcondition of `StrongConnect` we learn *data.Valid* is maintained. When we prove that `StrongConnect` maintains invariant $S_1$ we conclude that invariant $S_1$ holds.

We need to prove that invariant $S_1$ is maintained by a call of method `StrongConnect`$(u)$. By *PreconditionStrongConnect* we conclude that invariant $S_1(data)$ holds at the start of the `StrongConnect` method. There are four update sections, we need to prove that these sections maintain invariant $S_1$.

The first update to the global state *data* is the *AddNewNode* section, line 3 to 6. This section adds a node $u$ to *data.stack*, *data.disc* and *data.low*. The lemma *AddNewNodeDataValid* contains a proof which states that the invariant group *data.Valid* is maintained by the *AddNewNode* section. The invariant group *data.Valid* contains the invariant $S_1$. The lemma *AddNewNodeDataValid* uses sub-lemmas to create a stable proof. A sub-lemma contains a proof that one invariant is maintained. We create a lemma which proves that invariant $S_1$ is maintained by the *AddNewNode* section. This lemma proves that the invariant $S_1(data)$ holds, given that invariant $S_1(AddNewNode(data, u))$ holds. Let $v$ be a node in the $AddNewNode(data, u).stack$. We distinguish two cases.

- Case 1 $v = u$: By definition $u$ is connected to itself. Furthermore, the section *AddNewNode* sets $low[u]$ and $disc[u]$ to the same value. From this we can conclude that invariant $S_1$ holds for node $v$ if $v$ is equalt to $u$.
- Case 2 $v \neq u$: When $v \neq u$ then $v$ is in the old *data.stack*. From invariant $S_1(data)$ it follows that there exists a node $w \in data.stack$ which is connected to $v$ and $data.low[v] = data.disc[w]$. The section *AddNewNode* does not remove any nodes from the *stack* and the section *AddNewNode* maintains all values from *data.disc* and *data.low*. From this we can conclude that $w \in AddNewNode(data, u).stack$ and $AddNewNode(data, u).low[v] = AddNewNode(data, u).disc[w]$. Again, we conclude that invariant $S_1$ holds for $v$.

After this we prove that invariant $S_1$ is maintained in the first loop. There are three paths through the loop. We need to prove that invariant $S_1$ is maintained in all three cases. Let $v$ be the successor of node $u$ which is being processed and let *data* be the state at the start of an iteration. The first case is that $v$ is not in *data.low*. When $v \notin data.low$ then we first start with a recursive call `StrongConnect`$(v)$. From the postcondition of `StrongConnect`$(v)$ we conclude that *postData.Valid* and thus invariant $S_1(postData)$ holds where *postData* is the state after the call `StrongConnect`$(v)$ has completed.

After this the recursive call section *UpdateFromLow*, line 7, updates the state. This section sets the value of $low[u]$ to the minimum of itself and $low[v]$. The lemma *UpdateFromLowDataValid*, line 13, proves that $UpdateFromLow(postData, u, v).Valid$ holds. This lemma uses a separate lemma to prove that invariant $S_1$ is maintained. When $postData.low[u] \leq postData.low[v]$ then *postData* remains unchanged so invariant $S_1$ trivially holds. In the other case when

$postData.low[u] > postData.low[v]$ then $low[u]$ is set to $low[v]$. Let $w$ be a node in the *stack*. We distinguish two cases:

- Case 1 $w \neq u$: If $w \neq u$ then $w$ is connected to a node $x$ and $preData.low[w] = preData.disc[x]$. Since $preData.disc$, $preData.stack$ and $preData.low[w]$ are not updated invariant $S_1$ is maintained.
- Case 2 $w = u$: If $w = u$ then $u$ is connected to $v$ and $UpdateFromLow(data, u, v).low[u] = UpdateFromLow(data, u, v).low[v]$. Since $v \neq u$ we know that $v$ is connected to a node $x$ such that $x$ is in the *stack* and $low[v] = disc[x]$. We know that $low[u] = low[v] = low[x]$ and $u$ is connected to $x$ so invariant $S_1$ is maintained.

The second path through the loop is the case that the new neighbour $v$ is in *low* and $v$ is in *stack*. In this branch *data* is updated by the *UpdateFromDisc* function at line 19. The lemma *UpdateFromDiscDataValid* ensures that *UpdateFromDisc* maintains *data.Valid*. This lemma uses a lemma which proves that *UpdateFromDisc* maintains invariant $S_1$. This proof is rather trivial. If $data.low[u] \leq data.disc[v]$ then *data* is unchanged and otherwise $u$ is connected to $v$ and $low[u] = disc[v]$.

The last path through the loop is the case that $v$ is not in *stack* and in *low*. In this case *data* is unchanged and invariant $S_1$ is trivially maintained. This concludes the proof that the $S_1$ is maintained in the loop.

After the loop the state *data* is only updated if $disc[u] = low[u]$. If *data* remains unchanged then invariant $S_1$ is maintained. When $disc[u] = low[u]$ then the *PopFromStack* section, from line 25 to line 31, updates the state. The invariant *PopFromStackDataValid*, from line 23, proves that the section *PopFromStack* maintains invariant *data.Valid*. This lemma proves that invariant $S_1$ is maintained in a separate lemma. Let $v$ be a node in $PopFromStack(data, |\mathbf{old}(data).stack|).stack$. Here $\mathbf{old}(data)$ is the state at the art of `Strongconnect`$(u)$. We know that $v$ is also contained in *data.stack*. By invariant $S_1(data)$ we know that there exists a node $w$ in *data.stack* such that $w$ is connected to $v$ and $data.low[v] = disc[w]$. By invariant $S_0$ and $S_3$ we conclude that $w$ is located below $v$ in the stack. So $w$ is contained in $PopFromStackDataValid(data, |\mathbf{old}(data).stack|)$ and invariant $S_1$ is maintained. This concludes the proof that invariant $S_1$ is maintained.

# Chapter 7

# Proof of the iterative Tarjan's algorithm

In chapter 6 lemmas and invariant groups where introduced. These lemmas and invariants are used to verify the recursive version of Tarjan's algorithm. We want to reuse these lemmas and invariant groups to verify the iterative algorithm. This is possible because the recursive and iterative algorithm share a lot of similarities. The only one difference between the iterative and the recursive version is that the recursive algorithm automatically keeps track of the progress of every node by using recursion whereas the iterative algorithm manually stores the progress of all nodes in a new stack *work*.

The difference between the two versions of the algorithm is outlined in the following example. Let $u$ be a node which is being processed by the Tarjan's algorithm and let $v$ be the $j^{th}$ successors of $u$. Let the successor node $v$ be a node which has not yet been processed so $v \notin low$. Node $u$ cannot be completely processed until node $v$ has been processed. This is why the algorithm needs to postpone processing node $u$ until node $v$ has been processed. Both algorithms use a different approach to switch from processing node $u$ to node $v$. The recursive algorithm simply performs a recursive call `StrongConnect(v)` to switch from node $u$ to node $v$. Once the recursive call has completed the algorithm continues processing node $u$. Once the algorithm has completed then the algorithm automatically knows that $j$ neighbours of $u$ have been checked and that node $v$ has been processed. The iterative algorithm switches from node $u$ to node $v$ manually. The iterative algorithm pushes an entry with the process of node $u$ on the stack *work* and the algorithm then adds a new entry for node $v$ on top. These entries only contain the node which is being processed $u$ and the amount of processed neighbours $j$. The iterative algorithm retrieves one entries from the top of the stack *work* and the algorithm continues processing the entry. Once the node has completely been processed the algorithm retrieves the next node from the stack *work*. This means that the entry with node $u$ will be retrieved from the stack once node $v$ has completely been processed. This results in the same behaviour as the iterative algorithm.

Beside this difference the algorithms are the same. Both algorithms process nodes in the exact same order and the algorithms use the same main algorithm which calls `StrongConnect`. Because the main algorithm remains unchanged we can directly reuse the proof from the recursive algorithm as long as we don't change the post condition from the `StrongConnect` helper function. This is why we can reuse the lemmas from the recursive algorithm. However there is one problem, the proof of the recursive algorithm `StrongConnect`($u$) only reasons about the node $u$. In the iterative version of `StrongConnect`($u$) we need to reason about node $u$ and all unprocessed nodes which are reachable from $u$. There are three concrete problems:

1. The first problem is that the proof of the recursive algorithm uses old states of global variables. In a call `StrongConnect`($u$) the recursive proof refers to the old states **old**(*data*) and **recStart**(*data*). The state **old**(*data*) is the state at the start of the call and the state **recStart**(*data*) is the state before a recursive call. These old states are easy to reference in

the recursive proof since the recursive call is replaced by an assumption which states that the postcondition of the recursive call holds. These old states are not easy to reference in a proof of the iterative algorithm. In the iterative algorithm we require a different approach to maintain the **old** state and **recStart** state for any node in the *work* stack.

2. The second problem is that in order to prove a node $u$ is processed correctly we need to maintain an invariant *LoopInvariant*. This invariant states that $j$ successors of $u$ have been checked. Once all neighbours have been checked then we can create a new component and prove that the node has successfully been processed. In the recursive algorithm the invariant *LoopInvariant* is maintained for only one node. This invariant *LoopInvariant* is maintained in the successor loop, line 5 to 10 from algorithm 2. A recursive call can be performed in this loop, line 7. The recursive proof replaces this recursive call by a proof obligation that the precondition of `StrongConnect` holds and an assumption that the post-condition of `StrongConnect` holds. We cannot make this assumption in the iterative version of `StrongConnect`. In the iterative version of `StrongConnect` we need to prove that the invariant *LoopInvariant* is maintained for every node in the *work* stack. We can then prove that when a new successor is added to the *work* stack then the precondition holds and once the successor is processed then the postcondition holds. This allows us to prove that the assumption of the recursive algorithm holds and we can reuse the recursive proof.

3. The last problem is that the post conditions of `StrongConnect` needs to be verified. It is hard to define a loop invariant which is strong enough to derive the postcondition of the iterative `StrongConnect` method for the original input node.

The first and second problem can be solved by maintaining a second stack *ghostWork*. This stack will only be used for verification and will not influence the behaviour of the algorithm. The stack *ghostWork* will contain more information then *work*. An entry in *work* only contains a node and the amount of processed neighbours. An entry in *ghostWork* will also contain extra information like the relevant old states. Once problems one and two have been solved then we can write an invariant for the outer loop of the iterative algorithm which solves the third problem. These ideas will be outlined in detail in this chapter. The iterative algorithm outlined with lemmas and invariants can be found in algorithm 13 and 14

## 7.1 Storing old states global variables

We want to verify that the iterative version of Tarjan's algorithm is valid. If we reuse the specification of the recursive algorithm then we can reuse parts of the recursive proof. Let the precondition of `StrongConnect` be *PreconditionStrongConnect* and let the postcondition be *PostconditionStrongConnect*. The precondition *PreconditionStrongConnect* is defined in definition 6.7 and the postcondition *PostconditionStrongConnect* is defined in definition 6.6.

The main challenge with reusing the recursive proof is that the recursive call is replaced by an assert statement and an assumption. The recursive call `StrongConnect`$(v)$ is replace by an assert statement **assert** *PreconditionStrongConnect*$(data, v)$ and an assumption
**assume** *PreconditionStrongConnect*$(\mathbf{old}(data), v, data)$. The assert statement adds the proof obligation that the precondition of the recursive call holds and the assume statement introduces the assumption that the postcondition of the recursive call holds. Because the recursive call is replaced the recursive proof only reasons about one node and avoids proving that all successors nodes are processed correctly. The recursive proof only needs to keep track of two old states $\mathbf{old}(data), \mathbf{preData}(data)$ and one invariant *LoopInvariant*. Once all $j$ successors of a node $u$ have been processed then we can use the invariant *LoopInvariant*$(\mathbf{old}(data), data, u, j)$ to prove that the postcondition holds. The iterative version does not use recursion but mimics a recursive call by pushing two variables $(u, j)$ on the stack *work*. Here $u$ is the node which is being processed and $j$ is the amount of successors of $u$ which have been processed. Because we don't use recursion we cannot assume that all successor nodes are processed correctly. Next to this the old states are required for the proof are not stored.

In order to reuse the recursive proof we create a second second stack *ghostWork* as a *ghost* variable. A *ghost* variable is a variable which can only be used in the verification and which cannot influence the behaviour of the algorithm. The stack *ghostWork* stores all information which would be lost compared to the recursive algorithm. Using this stack *ghostWork* we can reuse the recursive proof in the iterative proof. We want to add one entry for every node which is being processed in this stack. One entry on the stack *ghostWork* will contain one node and all information which is required to prove that the node is being processed correctly. A new entry for a node $u$ being placed on the stack *work* that is similar to a recursive call. To reuse the recursive proof we have to prove that the predicate *PreconditionStrongConnect* holds. We will also add one entry with node $u$ on the stack *ghostWork*. After node $u$ is placed on the stack *work* then node $u$ is added to the global variables. We then want to use the entry of $u$ on the stack *ghostWork* to prove that the invariant *LoopInvariant* holds. Whenever a successor of $u$ is processed we want to prove that the *LoopInvariant* is maintained. This can be tricky if a successor has not been processed. Once all successors of $u$ have been processed we prove that *PostconditionStrongConnect* holds for node $u$. This postcondition can then be used in the proof of the node below $u$ in the *work* stack. Note that the node below $u$ in the stack is a predecessor of node $u$ which is waiting for node $u$ to be processed.

### 7.1.1 Defining Work

In this section we will define a new type *Work*. Every entry in the stack *ghostWork* will be of type *Work*. The type *Work* stores all information which is required to prove that one node is processed correctly. This proof uses the same lemmas and invariants from the recursive proof. The type *Work* has fields which stores all variables and old states which are required in the recursive proof. Next to this the type *Work* stores a phase. A phase is related to the invariants from the recursive algorithm. The recursive algorithm uses 3 invariants, we create one invariant related to every phase. Next to this we require one last phase to handle the recursion. In section 7.2 we will introduce a loop invariant which allows us to derive invariants from the recursive algorithm from the phase of a node. We distinguish the following 4 phases:

1. *WorkRequested*: This phase occurs when a node should be processed but no progress has been made. This phase is similar to the initial state in the recursive `StrongConnect` algorithm. In this phase the invariant *PreconditionStrongConnect* , from definition 6.7, holds and the node has not been added to the state *data*.

2. *WorkEnded*: This phase occurs when a node has fully been processed. If a node is fully processed then all neighbours of the node have been checked. This phase is similar to the final state of the recursive `StrongConnect` algorithm. In this state the invariant *PostconditionStrongConnect* from definition 6.6 holds.

3. *WorkStarted*: This phase indicates that we are currently actively processing a node. In this state the invariant *LoopInvariant* from definition 6.4 holds.

4. *WorkRecurse*: This phase indicates that we can currently not continue processing a node until a successor of the node has been processed. In this phase we store that the invariant *LoopInvariant* held before we started to process a successor. In the recursive proof Dafny simplifies the algorithm to avoid this phase. The recursive call is replaced with the assumption that the postcondition of the recursive call holds. We cannot use this assumption in the iterative algorithm. Because of this nodes can be in a new phase which is not considered in the recursive proof. If a node $u$ is in this phase then the node $u$ is waiting for a successor node $v$ to be processed. Once this successor $v$ has fully been processed then the successor $v$ is in the *WorkEnded* phase and node $u$ is on top of the stack *ghostWork*. If node $v$ is in the *WorkEnded* phase then we are going to be able to derive that the invariant *PostconditionStrongConnect* holds for node $v$. This invariant *PostconditionStrongConnect* was obtained by an assumption in the recursive proof. The recursive proof then used a

lemma after the recursive call to prove that if *LoopInvariant* held before the recursive call then we can update $low[u]$ after which the invariant *LoopInvariant* holds for node $u$. We require this the phase *WorkRecurse* in the iterative algorithm to prove that the invariant *LoopInvariant* held before the "recursive call". After node $v$ has been processed then $low[u]$ is updated and node $u$ moves back to the *WorkStarted* phase.

We have now introduced the four phases. Nodes changes phases while being processed. A state diagram of the phases is shown in figure 7.1 Every node initially starts in the *WorkRequest* phase where the invariant *PreconditionStrongConnect* holds. Once the node is added to the global variables *data* then node moves to the *WorkStarted* phase. In this phase the invariant *LoopInvariant* holds. In the *WorkStarted* phase all successor nodes are checked. If a successor has already been processed then we can easily prove that the *LoopInvariant* is maintained and we stay in the *WorkStarted* phase. If the neighbour has not been processed then the node moves to the *WorkRecurse* phase. In this phase the node waits until it is removed from the *work* stack. Once the node is removed from the *work* stack then we know that the successor node is in the *WorkEnded* phase. Because of this we can prove that the *LoopInvariant* holds again and we move back to the *WorkStarted* phase. Once all neighbours are processed then we finish processing the node by creating a new component or by leaving the node on the stack. After this the node has finished processing and we can prove that the invariant *PreconditionStrongConnect* holds and the node moves to the *WorkEnded* phase.

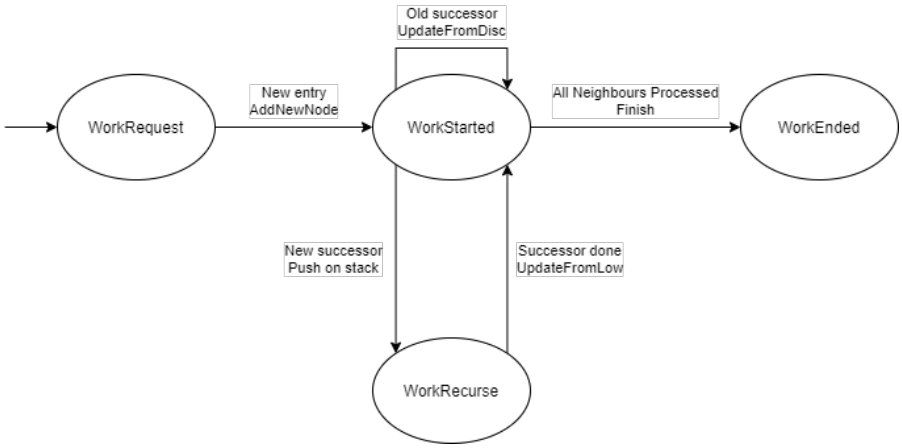


Figure 7.1: Phases when processing a Node

The exact definition of *Work* can be found in definition 7.1. The stack *ghostWork* should contain the same entries as the stack *work* but with more information. We enforce this using a loop invariant *WorkMatchesGhostWork*. This is an invariant of the outer loop from the iterative `StrongConnect` method. This invariant is defined in definition 7.2.

**Definition 7.1** (Work)**.** The type *Work* stores the progress of one node. This datatype stores information which is required to reuse the recursive proof. In the proof of the iterative version of Tarjan's algorithm we need to reason about multiple nodes. All nodes which are being processed are stored in a stack *ghostWork*. Every entry in this stack is of type *Work* and stores the progress of one node. This includes all values which are required to prove that all required invariants are maintained. A element of type *Work* can have 4 member types where each member type reflects a different phase. These member types are *WorkRequested*, *WorkStarted*, *WorkRecurse* and *WorkEnded*.

Every member type contains the fields $u, j, oldData$. Here $u$ is the node which is being tracked, $j$ is the amount of processed successors and *oldData* is the last state before we started processing node $u$. In the recursive proof the state *oldData* is called **old**(*data*). The member *WorkRecurse* also contains twp extra fields $v$ and *preData*. This member indicates that the node $u$ cannot be

processed until node $v$ has been processed. The field *preData* is the last state where $u$ was last processed. In the recursive proof this state was called **recStart**(*data*). The exact definition of *work* is shown below.
*work* $\equiv$

- $WorkRequest(u : Node, j : \mathbb{N}, oldData : TarjanData)$
- $WorkStarted(u : Node, j : \mathbb{N}, oldData : TarjanData)$
- $WorkRecurse(u : Node, j : \mathbb{N}, oldData : TarjanData, v : Node, preData : TarjanData)$
- $WorkEnded(u : Node, j : \mathbb{N}, oldData : TarjanData)$

**Definition 7.2** (WorkMatchesGhostWork)**.** The invariant $WorkMatchesGhostWork(work : seq\langle Node, \mathbb{N}\rangle, ghostWork : seq\langle Work\rangle)$ states that the *ghostWork* stack contains the same information as the *Work* stack. In other words both stacks have the same size and for every index $i$ it holds that $work[i].u = ghostWork[i].u$ and $work[i].j = ghostWork[i].j$. This gives us the following invariant:

$$WorkMatchesGhostWork(work, ghostWork) \equiv |work| = |ghostWork| \wedge$$
$$\forall i \in [0 \ldots |work|) : work[i].u = ghostWork[i].u \wedge work[i].j = ghostWork[i].j$$

### 7.1.2 Adding ghostWork to StrongConnect

In the previous section we have introduced the stack *ghostWork*. The stack *ghostWork* needs to be added to the `StrongConnect` method such that the loop invariant *WorkMatchesGhostWork* is maintained. An iteration of the outer loop only modifies the top element of the *work* stack. The top element is written to the variables $(u, w)$. Similarly we write the top element of the *ghostWork* stack to a new variable *ghostW* of type *Work*. Like *ghostWork* is a more inclusive version of *work*, *ghostW* is a more inclusive version of the pair of variables $(u, j)$. The variable *ghostW* is updated to match $(u, j)$. Next to this we update the member type of *ghostW* based on the phase. In this section we describe how *ghostWork* and *ghostW* are updated. The iterative version of `StrongConnect` annotated with the ghost variables is shown in algorithm 12. The ghost variables are insufficient to finish the proof. We will introduce invariants and lemmas to finish the proof after this section.

The stack *work* is updated three times in the iterative `StrongConnect` algorithm. The stack *ghostWork* also needs to be updated in the same locations:

1. Initially, at line 1 the stack *work* is set to $[(u_0, 0)]$. The ghost variable *ghostWork* is initially set to $[WorkRequest(u_0, 0, data)]$.
2. The second update occurs at the start of the the outer loop, line 6. Here the top of the stack *work* is popped from *work* and written to $(u, j)$. Similarly the top of *ghostWork* is popped from the stack and written to the variable *ghostW*.
3. The last update of *work* occurs from line 20 to 21. Here the progress from node $u$ is pushed on the stack and a new tuple $(v, 0)$ is added. The update from *ghostWork* is slightly more complex. The node $u$ cannot be processed until node $v$ has completely been processed. So node $u$ should be in phase *WorkRecurse*. This is why the variable *ghostW* needs to change from member type *WorkStarted* to *WorkRecurse*. After this *ghostW* is added to the stack *ghostWork* and a new entry *WorkRequest* is added on top of it.

The *ghostWork* stack is updated one more time at the end of the algorithm, line 39. At this point the variable $u$ has completely been processed. The top node on the *work* stack does was waiting on node $u$. The *low* value of the top node is updated and the node can be processed in the next iteration. Since the top node is no longer blocked the type of the top entry on the *ghostWork* stack is changed from *WorkRecurse* to *WorkStarted*.

The second variable *ghostW* also needs to be updated. We update *ghostW* in three different locations:

1. At line 11 it holds that *ghostW* is of type *WorkRequest*. Once node $u$ is added to *stack* then $u$ is no longer in the *WorkRequest* phase but $u$ is in the *WorkStarted* phase. Because of this we update *ghostW* to type *WorkStarted* at line 15.

2. The next update of *ghostW* occurs in the inner loop of the `StrongConnect` method. The variable $j$ is incremented in every iteration of the inner loop. The field *ghostW.j* is therefore also incremented at line 28.

3. The last update of *ghostW* occurs at line 38. Here the node $u$ has completely been processed. Therefore we change the state of *ghostW* to type *WorkEnded*.

**Algorithm 12** Partially annotated version of helper function StrongConnect

**Input**: $u$: An element of $Nodes$.

**precondition** $PreconditionStrongConnect(data, u_0)$

**postcondition** $PostconditionStrongConnect(\mathbf{old}(data), data, u_0)$

StrongConnect($u_0$):

1: **var** $work := [(u_0, 0)]$
2: **ghost var** $ghostWork := [WorkRequest(u_0, 0, data)]$
3: **ghost var** $ghostW := WorkRequest(u_0, 0, data)$
4: **while** $|work| > 0$ **do**
5: **invariant** $WorkMatchesGhostWork(work, ghostWork)$
6: $(u, j) := work[|work| - 1]$
7: $work := work[0 : |work| - 1]$
8: $ghostW := ghostWork[|ghostWork| - 1]$
9: $ghostWork := ghostWork[0 : |ghostWork| - 1]$
10: **if** $j = 0$ **then**
11: **var** $k := |data.disc|$ ▷ Section $AddNewNode(data, w.u)$
12: $data.disc[u] := k$
13: $data.low[u] := k$
14: $data.stack := data.stack +\!\!+ [u]$
15: $ghostW := WorkStarted(ghostW.u, ghostW.j, ghostW.oldData)$
16: $recurse := false$
17: **for** $i := j$ **to** $|successors(u)| - 1$ **do**
18: $v := successors(u)[i]$
19: **if** $v \notin data.low$ **then**
20: $work := work +\!\!+ [(u, i + 1)]$
21: $work := work +\!\!+ [(v, 0)]$
22: $ghostWork := ghostWork +\!\!+ [WorkRecurse(ghostW.u, ghostW.j + 1, ghostW.oldData, v, data)]$
23: $ghostWork := ghostWork\ +\!\!+ [WorkRequest(v, 0, data)]$
24: $recurse := true$
25: **break**
26: **else if** $v \in data.stack$ **then**
27: $data.low[u] := min(data.low[u], data.disc[v])$ ▷ Section $UpdateFromDisc(data, u, v)$
28: $ghostW.j := ghostW.j + 1$
29: **if** $\neg recurse$ **then**
30: **if** $data.low[u] = data.disc[u]$ **then**
31: $comp := [\,]$ ▷ Section $PopFromStack(data, |w.oldData.stack|)$
32: **while** $true$ **do**
33: $v := data.stack[|data.stack| - 1]$
34: $data.stack := data.stack[0 : |data.stack| - 1]$
35: $comp := comp +\!\!+ [v]$
36: **if** $v = u$ **then break**
37: $data.result := data.result +\!\!+ \{comp\}$
38: $ghostW := WorkEnded(ghostW.u, ghostW.v, ghostW.data)$
39: **if** $|work| > 0$ **then**
40: $v := u$
41: $(u, j) := work[|work| - 1]$
42: $data.low[u] := min(data.low[u], data.low[v])$ ▷ Section $UpdateFromLow(data, u, v)$
43: $ghostW := ghostWork[0 : |ghostWork| - 1]$ ▷ Change top of stack to $WorkStarted$
44: $ghostW := WorkStarted(ghostW.u, ghostW.j, ghostW.data)$
45: $ghostWork := ghostWork[0 : |ghostWork| - 1] +\!\!+ [ghostW]$

## 7.2 Incorporating invariants into Work

In section 7.1 a new data type *Work* and a new variable *ghostWork* was introduced. The variable *ghostWork* contains information on which nodes is being processed. The type *Work* contains information to prove that a node is processed correctly. An entry of type *Work* keeps track of the progress of one node. We distinguish 4 phases when we process a node, each phase is captured by a different member type of *Work*. These phases are *WorkRequest*, *WorkStarted*, *WorkRecurse* and *WorkEnded*. In this section we will first concretely define which invariants from the recursive algorithm should hold in every phase.

In order to relate the invariants from the recursive proof to the *Work* datatype we create an invariant *WorkValid* which accepts a *Work* entry as input. If this invariant holds then we can derive the invariants from the recursive algorithm based on the phase. The invariant *WorkValid* uses 4 smaller invariants: *ValidRequest*, *ValidStarted*, *ValidRecurse* and *ValidEnded*. These invariants define which invariant should hold in their respecitive phase. The invariant *WorkValid* simply matches the phases to the correct invariant. The invariant *WorkValid* is defined in definition 7.3. This invariant is an *opaque* predicate for performance reasons. Opaque predicates were explained in section 5.3.

**Definition 7.3** (WorkValid)**.** The invariant $WorkValid(w : Workdata : TarjanData)$ states that an element $w$ of type *Work* is valid for the given state *data*. A member of type *Work* tracks the progress of one node. We process a node in different phases. Depending on the phase different invariants should hold. We will define one unique invariant for every phase. The invariant *WorkValid* states an element $w$ of type *Work* is valid if the invariant corresponding to the member type of $w$ holds. This gives us the following invariant:
$WorkValid(w{:}Work,\ data{:}\ TarjanData) \equiv$

- $w.WorkRequest? \Rightarrow ValidRequest(w, data))$
- $w.WorkStarted? \Rightarrow ValidStarted(w, data))$
- $w.WorkRecurse? \Rightarrow ValidRecurse(w, data))$
- $w.WorkEnded? \Rightarrow ValidEnded(w, data))$

### 7.2.1 Exact Work Invariants

We want to relate the phases from *Work* to invariants from the recursive algorithm. Figure 7.2 shows the state diagram from section 7.1.1 annotated with the invariants and the lemmas from the recursive proof. The iterative `StrongConnect` algorithm annotated with lemmas is shown in algorithm 14.

The initial state *WorkRequest* indicates that a new node should be processed but no progress has been made. Let $w$ be of type *WorkRequest*, adding $w$ to the stack *ghostWork* indicates that node $w.u$ should be processed next. Adding $w$ to the top of the stack is similar to a call `StrongConnect(u)` in the recursive algorithm. At the start of the recursive algorithm the invariant *PreconditionStrongConnect* holds. We want $WorkValid(w, data)$ to imply *PreconditionStrongConnect*. Furthermore no progress should be made so $w.j$ should be 0 and $w.oldData$ should be *data*. This exact definition of $ValidRequest(w, data)$ is shown in definition 7.4.

From the *WorkRequest* phase a node can move to the *WorkStarted* phase. The *WorkStarted* phase indicates that a node is currently being processed. In this case the node has been added to the global state *data* and we are not waiting for a neighbour to finish processing. Let $w$ be of type *WorkStarted*. The entry $w$ states that node $w.u$ is currently being processed and $w.j$ of its successors have been processed. In the recursive algorithm this phase is similar to the successor loop, line 5 to 10 from algorithm 2. In this loop the invariant *LoopInvariant* should hold. We want *ValidStarted* to imply *LoopInvariant*. The invariant $ValidStarted(w, data)$ is defined in definition 7.5. In order to ensure that the invariant *ValidStarted* holds when we move from the *WorkRequest*

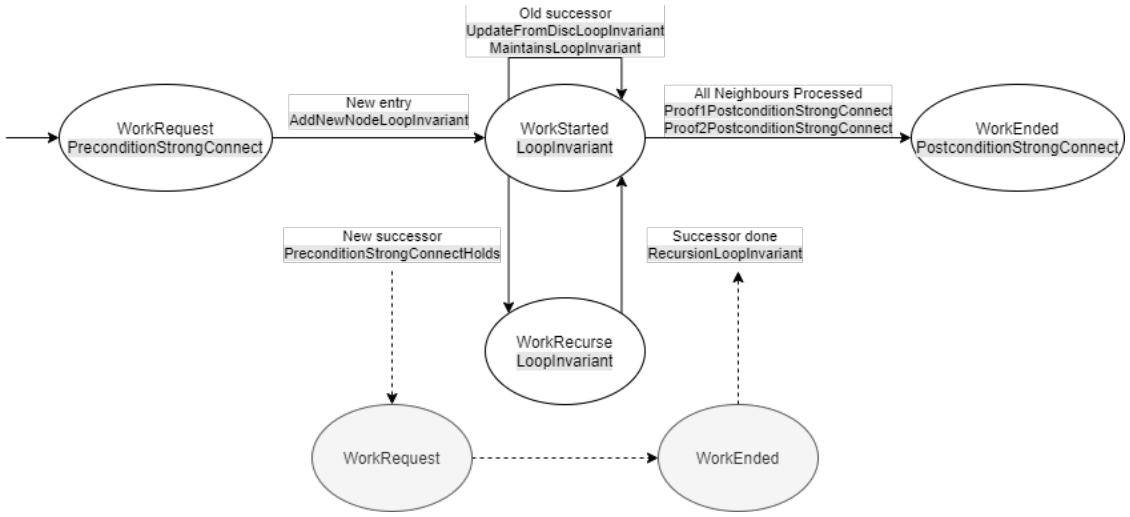


Figure 7.2: Annotated phases when processing a Node

phase to the *WorkStarted* phase we reuse the lemma *AddNewNodeLoopInvariant* from the recursive proof, definition C.5.

In state *WorkRequest* we can process a new neighbour and increment $w.j$. If the next neighbour is already part of a component then we can use the lemma *MaintainsLoopInvariant*, definition C.8, to prove that the invariant *ValidStarted* is maintained. If we encounter a node which is still in the stack *stack* then we can use the lemma *UpdateFromDiscLoopInvariant*, definition C.7, to prove that the invariant *ValidStarted* is maintained. Lastly we can process a neighbour which has not yet been processed. In this case we move to phase *WorkRecurse* and we create a new entry on the *ghostWork* stack of type *WorkRequest*. We can ensure that *WorkValid* holds using the lemma *PreconditionStrongConnect* holds.

The phase *WorkRecurse* can only be reached from state *WorkStarted*. The phase *WorkRecurse* indicates that a node cannot continue being processing until a successor node has been processed. Let $w$ be of type *WorkRecurse*. Here $w$ states that node $w.u$ is waiting for node $w.v$ to finish being processed. Node $w.v$ should be the $w.j - 1^{th}$ successor of $w.u$. After node $w.v$ has been processed then $low[w.u]$ is updated and $w$ moves back to phase *WorkStarted*. We want to prove that the invariant *ValidStarted* holds once node $w.u$ continues being processed. The invariant *ValidStarted* holds when we are able to prove that the invariant *LoopInvariant* holds. As discussed before, node $w.u$ waiting for node $w.v$ to be processed in the iterative algorithm is similar to a recursive call `StrongConnect`$(u)$ in the recursive algorithm. In the recursive algorithm the recursive call occurs at line 12 of the annotated recursive algorithm, algorithm 11. After the recursive call the lemma *RecursionLoopInvariant* is used to prove that the invariant *LoopInvariant* is maintained. We want to reuse this lemma in the iterative proof. The lemma *RecursionLoopInvariant* has the preconditions:

1. $LoopInvariant(w.oldData, w.recStart, w.v, w.j - 1)$
2. $PostconditionStrongConnect(w.recStart, data, w.v)$

In the iterative algorithm the update from low section occurs at line 42. In the iterative algorithm we know that node $w.v$ has an entry $w'$ which is in the phase *WorkEnded*. We are able to derive *PostconditionStrongConnect* from $ValidEnded(w', data)$. We want to derive the $LoopInvariant(w.oldData, w.recStart, w.v, w.j - 1)$ from $ValidRecurse(w, data)$. This gives us the definition of $ValidRecurse(w, data)$ as shown in definition 7.6.

The last phase *WorkEnded* can only be reached from *WorkStarted* once all successors have been processed. Let $w$ be in phase *WorkEnded*. As discussed before the invariant $ValidEnded(w, data)$ needs to imply that the invariant $PostconditionStrongConnect(w.oldData, w.u, data)$ holds. The

invariant *ValidEnded* has a straightforward definition and is defined in definition 7.7. We can directly reuse the proof of the recursive algorithm to prove that this invariant holds. If $low[w.u] = disc[w.u]$ then we can use lemma *Proof1PostconditionStrongConnect*, definition C.9, and otherwise we can use lemma *Proof2PostconditionStrongConnect*, definition C.10.

**Definition 7.4** (ValidRequest)**.** The invariant *ValidRequest*($w$ : *Work*, *data* : *TarjanData*) contains invariants which hold when a new node should be processed. Let $w$ be a work entry such that *ValidRequest*($w$, *data*) holds. This implies that $w.j = 0$ and $w.oldData = data$. Furthermore the invariant *PreconditionStrongConnect* should hold similar to the recursive algorithm, definition 6.7. This gives us the following invariant:
*Work.ValidRequest*($w$ : *Work*, *data* : *TarjanData*) $\equiv$

- $w.j = 0$
- $w.oldData = data$
- *PreconditionStrongConnect*($data, w.u$)

**Definition 7.5** (ValidStarted)**.** The invariant *ValidStarted*($w$, *Work*, *data* : *TarjanData*) contains all invariants which should hold when we are process a node. Let $w$ be a work entry such that *ValidRequest*($w$, *data*) holds. This means that we are currently processing node $w.u$ and the invariant *LoopInvariant* from definition 6.4 holds. This gives us the following invariant:

$$ValidStarted(w : Work, data : TarjanData) \equiv LoopInvariant(w.oldData, w.u, data, w.j)$$

**Definition 7.6** (ValidRecurse)**.** The invariant *ValidRecurse*($w$ : *Work*, *data*) contains all invariants which should hold when a node is waiting for another node to finish processing. This invariant maintains all invariants which used to hold when the node started waiting. Note that the iterative algorithm updates the value of $j$ when the process is postponed. The recursive algorithm only updates $j$ after the call continues. Because of this $j$ cannot be 0 and we need to process the $j - 1$ successor. Furthermore the current state *data* should be the state when $w.u$ was postponed. This gives us the following invariant:
*ValidRecurse*($w$ : *Work*, *data* : *TarjanData*) $\equiv$

- $data = w.recData$
- $0 < w.j \leq |successors(w.u)|$
- $successors(w.u)[w.j - 1] = w.v$
- *LoopInvariant*($w.oldData, w.u, w.preData, w.j - 1$)

**Definition 7.7** (Work.ValidEnded)**.** The invariant *ValidEnded*($w$ : *Work*, *data* : *TarjanData*) contains all invariants which should hold when a node is fully processed. This is the same as a postcondition of `StrongConnect`. This gives us the following invariant:

$$ValidEnded(w : Work, data : TarjanData) \equiv PostconditionStrongConnect(this.oldData, this.u, data)$$

### 7.2.2 Invariants of GhostWork

In section 7.1 the data type *Work* was introduced and a stack *ghostWork*. A variable $w$ of type *Work* tracks the progress of a node $w.u$. Next to this the invariant invariant *ValidWork*($w$, *data*) was introduced in section 7.2. This invariant implies that the state *data* contains the progress of a node $w.u$. Using the invariant *WorkValid* we can define a loop invariant which proves that all *Work* entries are waiting for the node above it in the *ghostWork* stack.

The invariant *GhostWorkIsValid*($ghostWork : seq < Work >, data : TarjanData$) states that the variable *ghostWork* properly stores the progress of all nodes. This invariant always holds if the stack *ghostWork* is empty. When the stack *ghostWork* is not empty then the top element $w$ of the

*ghostWork* stack should be valid for the current state *data*, ie the invariant $WorkValid(w, data)$ holds. Furthermore, since $w$ is on top of the stack node $w.u$ is either a new node which has not been processed or $w.u$ is a node which is no longer waiting for another node to be processed. We know that if $w.j = 0$ then $w$ is of type *WorkRequest* and if $w.j \neq 0$ then $w$ is of type *WorkStarted*. All elements which are not on top of the *work* stack are waiting for the node on top of the stack to be processed. So all entries in the tail of *ghostWork* have the member type *WorkRecurse* and all entries in the tail of the stack are waiting for the entry above it to complete processing. The invariant *GhostWorkIsValid* is defined in definition 7.8. This invariant uses the invariant *WorkIsCallFrom* which is defined in definition 7.9.

**Definition 7.8** (GhostWorkIsValid)**.** The invariant $GhostWorkIsValid(ghostWork : seq\langle Work\rangle, data : TarjanData)$ is a loop invariant of the main loop in `StrongConnect`. The variable *ghostWork* is valid if and only if the top of the stack is valid for the current state *data*. Furthermore the top of the stack has to be of type *WorkRequest* or *WorkRecurse*. The top of the stack is of type *WorkRequest* if and only if $ghostWork[0].j = 0$ and otherwise the top of the stack is of type *WorkStarted*. Any member below the first element on the stack *ghostWork* has to be of type *WorkRecurse*. Furthermore the invariant *WorkIsCallFrom* has to hold between any two consecutive entries in the stack. The invariant *WorkIsCallFrom* is defined in definition 7.9. This gives us the following definition:
$GhostWorkIsValid(ghostWork, data) \equiv ghostWork \neq [\,] \Rightarrow \textbf{var}\ w = ghostWork[0]$

- $WorkValid(w, data)$
- $w.j = 0 \Rightarrow w.WorkRequest?$
- $w.j \neq 0 \Rightarrow w.WorkStarted?$
- $\forall w \in ghostWork[\ldots|work| - 1] : w.WorkRecurse?$
- $\forall i \in [0, |ghostWork| - 2] : WorkIsCallFrom(ghostWork[i], ghostWork[i + 1])$

**Definition 7.9** (WorkIsCallFrom)**.** The invariant $WorkIsCallFrom(wOld : Work, wNew : Work)$ requires *wNew* and *wOld* to be of type *WorkRecurse*. This invariant implies that *wOld* is waiting for the entry *wNew* to complete. This simply means that $wOld.v = wNew.u$, that the intial state of *wNew* is *wOld.preData* and that *wOld* was valid in the state *wOld.preData*. This gives us the definition:
$WorkIsCallFrom(wOld, wNew) \equiv$

$$wOld.u = wNew.v \wedge wOld.preData = wNew.oldData \wedge ValidWork(wOld, wOld.preData)$$

### 7.2.3 Proof GhostWorkIsValid

The invariant *GhostWorkIsValid* allows us to prove that once a node has been processed then the invariant *PostconditionStrongConnect* holds. In order to prove that the loop invariant *GhostWorkIsValid* is maintained we reuse the recursive proof. It can be hard to see how the invariant *GhostWorkIsValid* can be verified and how it relates to the recursive proof. In this section we will outline the proof of *GhostWorkIsValid* and we will show where the lemmas from the recursive proof are used. We will omit the input arguments of the invariants and lemmas to improve readability. All lemmas and invariants including the input arguments are shown in the fully annotated algorithm 14. Verifying that the invariant *GhostWorkIsValid* is maintained requires proving that the invariant *WorkValid* is maintained for variable *ghostW*. Proving the variable *ghostW* is maintained depends on the phases. Remember that the relation between phases and the recursive lemmas and invariants is visualised in figure 7.2.

At the start of the partially annotated iterative algorithm, algorithm 12, the *ghostWork* stack is initially set to $WorkRequest(u_0, 0, data)$ at line 2. The invariant *PreconditionStrongConnect* holds for the initial node $u_0$ by the specification of `StrongConnect`. Using this we can easily prove that *GhostWorkIsValid* holds before the main loop.

Next we prove that the main loop maintains the loop invariant *GhostWorkIsValid*. At the start of the loop the top element is popped from the stack *ghostWork* and written to *ghostW*, line 8 to 9. From the invariant *GhostWorkIsValid* we learn that this top element is of member type *WorkRequest* if $j = 0$. If $j \neq 0$ then the element is of type *WorkStarted*. Furthermore, *GhostWorkIsValid* states that the invariant *WorkValid* holds for the top element. We want to prove that *WorkValid* is maintained for variable *ghostW* during the entire iteration.

If $j = 0$ then the iterative algorithm adds a node to the global state *data* in the section of code from line 11 to 14. This section is equal to the section of code *AddNewNode* from the recursive algorithm. After this *ghostW* is changed to the state *WorkStarted* at line 15. The invariant *WorkValid* is maintained if we can prove that the invariant *LoopInvariant* holds. The lemma *AddNewNodeLoopInvariant* from the recursive algorithm proves that the following Hoare triple holds:

$$\{PreconditionStrongConnect\}\, AddNewNode\{LoopInvariant\}$$

Using this lemma we can prove that the invariant *WorkValid* is maintained. After this it follows that *ghostW* is in phase *WorkStarted* for any value of $j$.

After this the iterative algorithm enters the successor loop from line 17 to 28. This loop is similar to the successor loop of the recursive algorithm. There are three possible paths trough the loop. If the next successor $v$ is in *data.stack* then the loop updates $low[u]$ based on $disc[v]$, line 27. This update is the same as the *UpdateFromDisc* from the recursive algorithm. The lemma *UpdateFromDiscLoopInvariant* from the recursive algorithm proves that the following Hoare triple holds:

$$\{LoopInvariant\}\, UpdateFromDisc\{LoopInvariant\}$$

Using this lemma we can prove that the invariant *WorkValid* is maintained for *ghostW*.

The second case is that $u$ is in *low* but not in *disc*. In this case $j$ is incremented but *data* is not changed. The recursive lemma *MaintainsLoopInvariant* proves that the invariant *WorkValid* is maintained in this case.

The third and last case is the most challenging case. In this case $v$ is not in *data.low*. In this case a recursive call is mimicked by placing $(u, i + 1)$ and $(v, 0)$ on the *work* stack after which the algorithm breaks out of the inner loop and ends the iteration of the main loop, line 20 to 25. Since this ends the iteration of the main loop we need to prove that the invariant *GhostWorkIsValid* holds. We know that the invariant *WorkValid* holds for *ghostW* and since *ghostW* is in the phase *WorkStarted* we can conclude that *LoopInvariant* holds at line 20. Since *LoopInvariant* holds we can prove that the invariant *WorkValid* holds for the new entry *WorkRecurse* on the stack *ghostWork*. The second entry on the stack *ghostWork* is the new entry $WorkRequest(v, 0, data)$. The invariant *WorkValid* holds for this entry if the invariant *PreconditionStrongConnect* holds. The lemma *PreconditionStrongConnectHolds* from the recursive algorithm proves that this invariant can be derived from *LoopInvariant*. Since the invariant *WorkValid* holds for both new entries on the *ghostWork* stack we can easily prove that the invariant *GhostWorkIsValid* is maintained.

Unfortunately we are not done yet. The successor loop can also terminate naturally once all successors have been checked. In this case the variable *recursing* is *false* and the iterative algorithm enters the last section starting on line 30. Here we know that the invariant *LoopInvariant* holds and all neighbours of $u$ have been checked. The iterative algorithm now finishes processing node $u$, line 30 to 38. This code is identical to the end of the recursive algorithm. Using lemmas *Proof1PostconditionStrongConnect* and *Proof2PostconditionStrongConnect* from the recursive algorithm we can prove that the invariant *PostconditionStrongConnect* holds for $u$ at line 38. Because of this we can change *ghostW* to the final phase *WorkEnded* such that the invariant *WorkValid* is maintained.

When *work* is empty then the iteration of the main loop is finished and the loop terminates. Since *ghostWork* has the same size as *work* it follows that *ghostWork* is also empty so the invariant *GhostWorkIsValid* trivially holds. In the other case that *work* is not empty then we need to peek at the top element from the *work* stack and update *data.low*, line 39 to 42. This section is equal the *UpdateFromLow* section from the recursive algorithm. The lemma *UpdateFromLowLoopInvariant*

is a complicated lemma from the recursive algorithm which proves that the following Hoare triple holds:

$$\{LoopInvariant(oldData, preData, u) \land PostconditionStrongConnect(preData, v, data)\}$$

$$UpdateFromDisc(data, u, v)\{LoopInvariant(oldData, data, u)\}$$

We want to change the top element of the stack *ghostWork* to phase *WorkStarted*. In order to prove that the invariant *GhostWorkIsValid* holds we need to prove that *WorkValid* holds after we change the top element to phase *WorkStarted*. This invariant holds if and only if *LoopInvariant* holds for the current state *data*. In order to prove this we need use the lemma *UpdateFromLowLoopInvariant* at line 39. We know that the invariant *GhostWorkIsValid* held at the start of the main loop. Using this invariant we can derive that the top element is in the phase *WorkRecurse*, the invariant *WorkValid* holds and the top element is waiting for the element *ghostW* to be processed. From this we can derive that the predicate *LoopInvariant*(*oldData*, *preData*, *u*) holds. Next we have proven that *ghostW* is in phase *WorkEnded* and the invariant *WorkValid* holds. From this we can derive that the invariant *PostconditionStrongConnect*(*preData*, *v*, *data*) holds. Using these two invariants we can then call *UpdateFromLowLoopInvariant* and prove that *GhostWorkIsValid* is maintained. This covers the all paths trough the loop and proves that the loop invariant *GhostWorkIsValid* is maintained in every iteration of the main loop.

## 7.3 Proving the postcondition

In the previous sections we have defined ghost variables and invariants which allow us to incorporate the recursive proof in the proof of the iterative algorithm. Unfortunately, we still need to prove that the postcondition *PostconditionStrongConnect*(**old**(*data*), $u_0$, *data*) from the `StrongConnect` method holds. We need to define a loop invariant for the outer loop such that we can derive the postcondition once the outer loop terminates.

Let a work entry $w$ be *WorkEnded*($u_0$, |*successors*($u_0$)|, **old**(*data*)) where **old**(*data*) is the state at the start of the iterative `StrongConnect` call. The invariant *WorkValid*($w$, *data*) implies that the postcondition of `StrongConnect` holds. We want to prove that *WorkValid*($w$, *data*) holds when the outer loop terminates, ie. $work = [\,]$. Note that when $work = [\,]$ holds then by loop invariant *GhostWorkMatchesValid* , definition 7.2, $ghostWork = [\,]$ holds.

We use two observations to define a suitable loop invariant to prove the postcondition. The first observation is that the initial node $u_0$ is always at the bottom of the *work* stack. Using invariant *WorkMatchesGhostWork* we conclude that the bottom element in the *ghostWork* stack contains a *Work* entry $w$ such that $w.u = u_0$.

The second observation is that the variable *ghostW* always contains an entry *WorkEnded* with the last processed node if the size of *work* decreases. The proof of the invariant *GhostWorkValid* already proves that the invariant *WorkValid*(*ghostW*, *data*) is maintained. So if the *work* stack decreases in an iteration then *ghostW* is of type *WorkEnded* and using invariant *WorkValid* we can derive *PostconditionStrongConnect* for the node *ghostW.u*.

Using these two observations we know that if *work* is empty then $ghostW.u = u_0$ and *ghostW* is of type *WorkEnded* from which we can conclude that the postcondition of `StrongConnect` holds. This intuitive proof is captured in the loop invariant *GoalInvariant* from definition 7.10. The proof that this loop invariant is maintained takes some effort but is not difficult. The entire iterative version of the Tarjan's algorithm annotated with lemmas, variables and invariants is shown in algorithm 13 and 14. The iterative algorithm also contains a small inner loop. The loop invariant for this inner loop is defined in definition 7.11. This invariant looks complex but simply specifies the behaviour of the inner loop.

**Definition 7.10** (GoalInvariant)**.** The invariant *GoalInvariant*(*ghostWork* : *seq*⟨*Work*⟩, *ghostW* : *Work*, *oldData* : *TarjanData*, *data* : *TarjanData*) is a loop invariant of the outer loop of the iterative `StrongConnect` algorithm. The post condition of `StrongConnect` can be derived from this invariant once the outer loop terminates. The invariant *GoalInvariant* ensures that the bottom of the

stack *ghostWork* stores information about the original node $u_0$, that invariant *WorkValid* holds for *ghostW* and that when *ghostWork* is empty then *ghostW* is $WorkEnded(u_0, |successors(u_0)|, oldData)$. This gives us the invariant:
$GoalInvariant(ghostWork, ghostW, oldData, data) \equiv$

- $data.Valid()$
- $ValidWork(ghostW, data)$
- $[\,] = ghostWork \Rightarrow ghostW = WorkEnded(u_0, |successors(u_0)|, oldData)$
- $[\,] < ghostWork \Rightarrow ghostWork[0].u = u_0 \wedge ghostWork[0].oldData = oldData$

**Definition 7.11** (InnerLoopInvariant)**.** The invariant $InnerLoopInvariant(work : seq\langle(Node, \mathbb{N})\rangle, oldGhostWork : seq\langle Work\rangle, ghostWork : seq\langle Work\rangle, oldW : Work, ghostW : Work, data : TarjanData, recurse : \mathbb{B})$ is a loop invariant for the inner loop of the iterative `StrongConnect` function. This invariant is simply the specification of the effect of the inner loop. This specification allows us to prove that the invariants of the outer loop are maintained. The input variables *oldGhostWork* and *oldGhostW* are the values *ghostWork* and *ghostW* at the start of the loop.
$InnerLoopInvariant(work, oldGhostWork, ghostWork, oldW, w, data, recurse) \equiv$

- $data.Valid()$
- $ghostW.WorkStarted?$
- $ValidWork(data, ghostW)$
- $0 \leq j \leq |successors(ghostW.u)|$
- $oldGhostW.u = oldGhostW.u \wedge oldGhostW.oldData = ghostW.oldData$
- $\neg recurse \Rightarrow oldGhostWork = ghostWork[\ldots|ghostWork| - 1]$
- $recurse \Rightarrow$ **var** $v = successors(ghostW.u)[ghostW.j];$
  $oldGhostWork = ghostWork[\ldots|ghostWork| - 1] +$
  $[StartToRec(ghostW.u, ghostW.j{+}1, ghostW.oldData, v, data)] + [WorkStarted(v, 0, data)]$
- $recurse \Rightarrow WorkIsValid(ghostWork, data)$

**Algorithm 13** Tarjan's Strongly Connected Component Algorithm with proof constructs

**Input**: $G = (V, E)$: A graph with nodes $V$ and edges $E$.
**Output**: *result*: A set containing all strongly connected components of the graph.
**postcondition** *data.result.Valid*()
TARJAN($G$):
1: **var** $data := TarjanData(DiscLow(map[\,], map[\,], Result(\{\,\}), stack([\,]))$
2: **for** $u \in V$ **do**
3: **invariant** $MainLoopInvariant(data, u)$
4: **if** $u \notin low$ **then**
5: STRONGCONNECT($u$)
6: **return** *data.result*

**Algorithm 14** Fully annotated version of helper function StrongConnect

**Input**: $u$: An element of $Nodes$.

**precondition** $PreconditionStrongConnect(data, u_0)$

**postcondition** $PostconditionStrongConnect(\mathbf{old}(data), data, u_0)$

StrongConnect($u_0$):

1: $work := [(u_0, 0)]$
2: $ghostWork := [WorkRequest(u_0, 0, data)]$
3: $ghostW := WorkRequest(u_0, 0, data)$
4: **while** $|work| > 0$ **do**
5: **invariant** $GoalInvariant(ghostWork, ghostW, data)$
6: **invariant** $GhostWorkIsValid(ghostWork, data)$
7: **invariant** $WorkMatchesGhostWork(work, ghostWork)$
8: $(u, j) := work[|work| - 1]$
9: **var** $work := work[0 : |work| - 1]$
10: **ghost var** $ghostW := ghostWork[|ghostWork| - 1]$
11: **ghost var** $ghostWork := ghostWork[0 : |ghostWork| - 1]$
12: **if** $j = 0$ **then**
13: **lemma** $AddNewNodeDataValid(data, u)$
14: **lemma** $AddNewNodeLoopInvariant(data, u)$
15: **var** $k := |data.disc|$
16: $data.disc[u] := k$
17: $data.low[u] := k$
18: $data.stack := data.stack +\!\!+ [u]$
19: $ghostW := WorkStarted(ghostW.u, ghostW.j, ghostW.oldData)$
20: $recurse := false$
21: **for** $i := j$ **to** $|successors(u)| - 1$ **do** ▷ **init**($ghostWork$), **init**(w)
22: **invariant** $InnerLoopInvariant(work, \mathbf{init}(ghostWork), ghostWork, \mathbf{init}(w), data, recurse)$
23: $v := successors(u)[i]$
24: **if** $v \notin data.low$ **then**
25: $work := work +\!\!+ [(u, i + 1)]$
26: $work := work +\!\!+ [(v, 0)]$
27: $ghostWork := ghostWork +\!\!+ [WorkRecurse(ghostW.u, ghostW.j + 1, ghostW.oldData, v, data)]$
28: **lemma** $PreconditionStrongConnectHolds(data, u, v)$
29: $ghostWork := ghostWork +\!\!+ [WorkRequest(v, 0, data)]$
30: $recurse := true$
31: **break**
32: **else if** $v \in data.stack$ **then**
33: **lemma** $UpdateFromDiscDataValid(data, u, v)$
34: **lemma** $UpdateFromDiscLoopInvariant(ghostW.oldData, data, u, v, v\_index)$
35: $data.low[u] := min(data.low[u], data.disc[v])$
36: **else**
37: **lemma** $MaintainsLoopInvariant(ghostW.oldData, data, u, v, v\_index)$
38: $ghostW.j := ghostW.j + 1$

```
39:  if ¬recurse then
40:      if data.low[u] = data.disc[u] then
41:          lemma PopFromStackDataValid(data, |ghostW.oldData.stack|)
42:          lemma Proof1RecursionInvariant(ghostW.oldData, data, u)
43:          comp := [ ]
44:          while true do
45:              v := data.stack[|data.stack| − 1]
46:              data.stack := data.stack[0 : |data.stack| − 1]
47:              comp := comp ++ [v]
48:              if v = u then break
49:          data.result := data.result ++ {comp}
50:      else
51:          lemma Proof2RecursionInvariant(ghostW.oldData, data, u)
52:      ghostW := WorkEnded(ghostW.u, ghostW.v, ghostW.data)
53:      if |work| > 0 then
54:          v := u
55:          (u, j) := work[|work| − 1]
56:          lemma UpdateFromLowDataValid(data, u, v)
57:          lemma RecursionLoopInvariant(ghostW.oldData, ghostW.preData, data, u, v)
58:          data.low[u] := min(data.low[u], data.low[v])
59:          ghostW := ghostWork[0 : |ghostWork| − 1]   ▷ Change top of stack to WorkStarted
60:          ghostW := WorkStarted(ghostW.u, ghostW.j, ghostW.data)
61:          ghostWork := ghostWork[0 : |ghostWork| − 1] ++ [ghostW]
```

## 7.4 Termination of the iterative algorithm

We have introduced a proof of correctness for the recursive and iterative version of Tarjan's algorithm. These proofs prove that the Tarjan's algorithm confirms with its specification given that the algorithm terminates. We have briefly introduced the proof that the recursive algorithm terminates in section 4.2.1. The recursive termination proof is rather trivial but unfortunately we cannot directly reuse the recursive termination proof. The iterative termination proof is a lot more involved.

We need to prove that the iterative algorithm terminates. The iterative algorithm does not use recursion so the iterative algorithm terminates if both loops terminate. The inner loop of the iterative algorithm iterates over all neighbours of a node. This loop trivially terminates and Dafny can prove this without any assistance. The outer loop terminates once the *work* stack is empty. Proving that the *work* stack will eventually be empty is not trivial. In order to prove that the outer loop terminates we define a termination measure. A termination measure consists of one or more expressions which all have a lower bound. In the recursive proof the termination measure only consists of one expression. If a termination measure consists of one expression then this expression needs to decrease in every loop iteration. There should also be a constant non zero lower bound on the decrease of the expression. From the facts that the expression has a minimum and the expression decrease it follows that the algorithm has to terminate.

When a termination measure consist of multiple expressions then the proof is more complex. The expressions of a termination measure are ordered. If a termination measure consists of $e_1 \dots e_j$ expressions then in every iteration there has to exist an expression $e_i$ with $i \in [1, j]$ where the value of expression $e_i$ decreases, with a non zero minimum bound and the value of all expressions before $e_i$ don't increase. Note that the value of expressions after $e_i$ can increase. This termination measure is strong enough to ensure that the loop will eventually terminate. Lets say that in an iteration only the last expression $e_j$ decreases. Then the value of all other expressions has to remain the same. Since the expression $e_j$ has a lower bound eventually the value of expression $e_{j-1}$ or another expression before $e_j$ has to decrease. The value of expression $e_j$ can now increase. Eventually the expression $j$ cannot decrease again and the process repeats. This process cannot continue infinitely since eventually all expressions have to reach their minimum value. From this we can conclude that the termination measure ensures that an algorithm terminates.

We prove that the outer loop of the iterative algorithm terminates using the following termination measure:

$$\texttt{NodeLimit} - |low \cup Nodes(work)|,\ |work|$$

Here `NodeLimit` is the amount of nodes and the function *Nodes* returns all nodes which have an entry in *work*. The first expression of the termination measure is equal to the amount of nodes which are neither in *low* nor *work*. The second expression is simply the size of *work*. Since there exist a finite amount of nodes both expressions trivially have a lower bound. We still need to prove that this termination measure correctly decreases. In every iteration of the outer loop there are two cases. The first case is that a node finishes processing and the second case is that a new node is added to the *work* stack.

In the first case let $u$ be the node which has been processed. At the start of the iterative algorithm the node $u$ is removed from the stack *work* at line 8. The algorithm does not add any nodes to the *work* stack in this case. From this we can conclude that the size of *work* decreases. The algorithm does not remove any other nodes from *work* and no nodes are removed from *low*. Furthermore the algorithm guarantees the node $u$ is in *low* at the end of the iteration. Since no nodes are removed from the set $low \cup Nodes(work)$ it follows that the value of the first expression does not increase. From this we conclude that the termination measure correctly decreases in this scenario.

The second case is that a new node $v$ is added to the *work* stack. The size of $|work|$ increases so we need to prove that the expression $\texttt{NodeLimit} - |low \cup Nodes(work)|$ decreases. From line 24 we learn that the new node $v$ is not in *low*. If we prove that $v$ was not in $Nodes(work)$ then the termination measure decreases. We can actually prove this without using new invariants just

using *GhostWorkIsValid*. The idea behind this proof is that it follows from *ValidRecurse* that all nodes in *work* are in *low* and since $v \notin low$ it follows that $v \notin Nodes(work)$. This concludes the termination proof of the iterative algorithm.

# Chapter 8

# Conclusions

In this paper we have presented a formal proof for two versions of Tarjan's algorithm. These algorithms are verified using proof annotations which are specified directly in terms of the program variables. The proof annotations of the recursive algorithm are based on a pre-existing proof. The end goal of this paper is to verify that the iterative version of Tarjan's algorithm is correct. In order to verify this algorithm we first verify the recursive algorithm and we then extend the proof to the iterative algorithm. Unfortunately, it is not possible to reuse the pre-existing proof of the recursive algorithm. The pre-existing proof requires a lot of effort from the Dafny verifier. The verification of the pre-existing proof can behave unexpectedly and fail to verify after minor changes to the proof. When this occurs we say that a proof is unstable. It is not feasible to scale the unstable pre-existing recursive proof to the iterative algorithm. This is why we have introduced techniques to ease the verification process for Dafny. These techniques allow us to stabilize proofs. A link to the Dafny proof of both versions of Tarjan's algorithm can be found in appendix A. Next to this appendix A also contains the Dafny code of the pre-existing proof and statistics obtained from running the Dafny verification.

The recursive and iterative proof are both verified using the pre-release of Dafny version 3.0.0. The pre-existing proof cannot be verified in Dafny version 3.0.0, this proof can be verified in Dafny version 2.3.0. The statistics of the new proofs are shown in table 8.1. The statistics of the pre-existing verification are shown in table 8.2. These tables show the amount of proof obligations and the verification time in seconds per file. Next to this Dafny verifies a proof in two steps. In the first step, the well formed step, Dafny verifies that the proof is well formed. In this step Dafny checks if all definitions are valid and if all variables have the correct type. Dafny tends to generate a lot of proof obligations in this steps but Dafny is almost always able to verify these obligations without assistance from the developer. The next step is the implementation step. Dafny verifies that all proofs and specifications are correct in this step. Dafny usually requires assistance from the developer in this step. The tables 8.1 and 8.2 also show the amount of proof obligations and the total verification time for each step.

| File name | Total o | Total t | Form o | Form t | Impl o | Impl t |
|---|---|---|---|---|---|---|
| graph_defs | 222 | 5.2 | 62 | 3.1 | 160 | 2.1 |
| iterative_defs | 136 | 7.4 | 136 | 7.4 | 0 | 0.0 |
| iterative_lemmas | 172 | 5.6 | 39 | 2.4 | 133 | 3.1 |
| IL_successorLoop | 257 | 16.1 | 34 | 2.6 | 223 | 13.6 |
| IL_termination | 55 | 2.8 | 13 | 1.6 | 42 | 1.2 |
| iterative_main_abstract | 260 | 17.1 | 5 | 1.4 | 255 | 15.7 |
| iterative_main_exact | 323 | 71.4 | 5 | 1.7 | 318 | 69.8 |
| recursive_defs | 303 | 14.6 | 242 | 12.6 | 61 | 2.0 |
| recursive_lemmas | 275 | 5.5 | 73 | 1.8 | 202 | 3.7 |
| RLDV_AddNewNode | 128 | 11.6 | 11 | 0.9 | 117 | 10.7 |
| RLDV_PopFromStack | 312 | 13.1 | 61 | 2.3 | 251 | 10.8 |
| RLDV_UpdateFromDisc | 448 | 117.3 | 117 | 20.6 | 331 | 96.7 |
| RLDV_UpdateFromLow | 329 | 68.2 | 99 | 8.0 | 230 | 60.2 |
| RLLV_UpdateFromLow | 527 | 27.5 | 92 | 2.3 | 435 | 25.1 |
| RL_successorLoop | 455 | 12.3 | 83 | 2.7 | 372 | 9.5 |
| recursive_main_abstract | 215 | 2.6 | 6 | 1.0 | 209 | 1.6 |
| recursive_main_exact | 215 | 2.7 | 6 | 1.0 | 209 | 1.7 |
| **Total recursive** | **3207** | **275.4** | **790** | **53.4** | **2417** | **222.0** |
| **Total iterative** | **4632** | **401.2** | **1084** | **73.7** | **3548** | **327.5** |

[o] Amount of proof obligations.
[t] Verification time in seconds
[Form] The well formed step in the verification process
[Impl] The implementation step in the verification

Table 8.1: Proof obligations and verification time Dafny implementation

| File name | Total o | Total t | Form o | Form t | Impl o | Impl t |
|---|---|---|---|---|---|---|
| soundness-completeness | 1178 | 59.1 | 370 | 3.4 | 808 | 55.7 |
| termination | 161 | 1.1 | 26 | 0.5 | 135 | 0.5 |
| graph_algorithm | 669 | 4.0 | 238 | 2.5 | 431 | 1.4 |
| **Total** | **2008** | **64.1** | **634** | **6.4** | **1374** | **57.7** |

[o] Amount of proof obligations.
[t] Verification time in seconds
[Form] The well formed step in the verification process
[Impl] The implementation step in the verification process

Table 8.2: Pre-existing proof obligations and verification time

## 8.1 Comparison pre-existing proof to new recursive proof

When we compare the statistics from the pre-existing proof with the new recursive proof then we first note that the total amount of proof obligations have increased from 2000 to 3000. Furthermore, the pre-existing proof took a total of 1 minute to verify whereas the new proof takes 5 minutes. This increase in proof obligations and verification time was expected. From table 8.1 and 8.2 we learn that the amount of proof obligations have increased significantly in the implementation step. The proof obligations have not significantly increased in the well formed step however the verification time has increased drastically.

The increase in total verification time of the implementation step was expected. The new proof creates a stable proof by separating the proof into different lemmas. Splitting a monolithic proof into multiple lemmas introduces overhead. Every lemma will have a smaller execution time and less proof obligations then the monolithic proof. However, the sum of all proof obligations and the total verification time will be larger. Because every lemma has less proof obligations the proof using smaller lemmas will be more stable then the large proof. Next to this the increase in total verification time is not a problem since we can verify and debug every lemma separately. To compare the new recursive proof with the pre-existing proof, the new proof uses a total of 87 lemmas whereas the pre-existing proof only uses 19 lemmas. This explains why the implementation step takes a lot longer to verify. Next to this the new proof also experiences overhead from revealing opaque predicates.

The increase in verification time from the well formed step was initially unexpected. Without aggregating all statistics this increase is not noticeable since the increase is spread over all lemmas. The implementation step takes 54 seconds in the new proof while the pre-existing proof only took 6 seconds. Next to this the new proof has 790 proof obligations while the pre-existing proof has 634 proof obligations. The small increase in proof obligations was expected however the large increase in verification time was not. This increase in verification time turns out to be a side-effect of opaque predicates. We will often need to reveal a small fact from an opaque predicate to prove that a specification is will formed. Manually revealing an opaque predicate creates an overhead of about one second per reveal. This increase is not really noticeable however it adds up over time. As concrete example of why we need to reveal predicates in the well formed step let $P$ be an opaque predicate with definition $u \in low \wedge P'$ and let $L(x)$ be a lemma with precondition $P \wedge low[u] = x$. The precondition is only well formed if $u$ is in $low$. We do not want to reveal $P$ every time we call lemma $L$ so we do not want to add $u \in low$ to the precondition. We can circumvent this problem by revealing $P$ when we verify that the precondition is well formed. This allows us to call $L$ without revealing $P$ but it does introduce some small overhead in the well formed step.

## 8.2 Proof of the recursive version Tarjan's algorithm

We created the proof of the recursive algorithm to make the proof as stable as possible. The critical step in the recursive proof is the verification of the `StrongConnect` method. We wanted to stabilize this step as much as possible. The pre-existing proof is unable to create a stable verification of this method. This is why the pre-existing proof needs to prove partial correctness by proving termination in a separate proof. The pre-existing proof is optimized to verify partial correctness of the `StrongConnect` method in 14.7 seconds with a total of 483 proof obligations on Dafny version 2.3.0. In comparison to this the new proof of the `StrongConnect` method can prove both correctness and termination in one stable proof. This new proof takes 1.3 seconds and has 175 proof obligations.

These numbers show a huge decrease in verification time and proof obligations. The new proof requires less than $\frac{1}{3}$ of the proof obligations then the pre-existing proof. Because of this the time required to verify the top level method decreased over 90%. The new proof of the `StrongConnect` method is extremely stable while the pre-existing proof is not. The impact of this change might not be clear when we just compare the verification time. When both proofs verify then the verification

time only differs a few seconds however the impact is far larger when the proof does not verify. If Dafny fails to verify some predicate in the new proof then Dafny will be able to identify the problem within a few seconds. If the pre-existing proof fails to verify a predicate then Dafny might take multiple minutes to find the problem or Dafny might just run out of resources. Next to this unlike the new proof the pre-existing proof will behave unexpectedly after minor changes. This is why we can scale the new recursive proof to verify the iterative algorithm. The statistics clearly show the impact of our techniques. The top level verification of the `StrongConnect` method has gone from a proof at the limits of Dafny's capabilities to a trivial proof. We expect that similar results can be obtained in other proofs.

Unfortunately, there is also one large disadvantage to our techniques. The new proof requires a lot more definitions and lemmas. The pre-existing proof only requires 19 lemmas whereas the new recursive proof requires 87 lemmas. Creating all these lemmas and definitions obviously requires a lot of extra effort from the developer. It was worth completely optimizing the recursive proof since we could reuse the lemmas and the gain from every lemma was large. It is quicker to verify a stable proof which consists of a lot of lemmas then an unstable proof with a few lemmas. However when a proof with few lemmas is stable then we do not recommend completely optimizing the proof unless you can reuse the lemmas in a different proof. We have incrementally applied these techniques to stabilize the iterative proof. We incrementally created a straightforward iterative proof and we only applied the techniques from chapter 5 when the iterative proof became unstable. As long as the proof was stable then there is no need to apply any techniques to further stabilize the proof. This saves a lot of extra effort from the developers. We generally recommend building a proof incrementally and only using the techniques when they are necessary.

## 8.3 Proof of the iterative version Tarjan's algorithm

The iterative version of Tarjan's algorithm can be verified in about 7 minutes and the proof has 4632 proof obligations. The iterative proof reuses all lemmas and predicates from the recursive algorithm which greatly simplifies the verification process of the iterative algorithm. However even with all the efficient lemmas from the recursive proof we still needed to use our techniques to create a stable iterative proof. We have once again used the techniques from chapter 5 to stabilize the recursive proof. However, unlike the recursive proof we have not used these techniques to decrease the verification time as much as possible. We have only applied these techniques to create a stable proof. The remaining verification of the proof still occurs in the `StrongConnect` method. The current verification has been stabilized enough to successfully verify the iterative algorithm and the verification stays stable even after minor changes. However, the iterative algorithm is nearing the limits of Dafny's capabilities. A minor change can have a large impact on the verification time.

A good example of the stability of the stability of the iterative proof is seen in the two files *iterative_main_abstract* and *iterative_main_exact*. These files both contain the same proof of the iterative version of Tarjan's algorithm. The difference is that in the abstract version all sections of code with lemmas have been replaced by functions whereas the exact file contains the complete algorithm. This abstract file is mainly useful for debugging purposes. Replacing sections of code by functions is more readable and slightly more efficient. Beside this the algorithms contain the same proof. Even though the difference is that small the abstract proof takes 17 seconds while the exact proof takes 71 seconds. This indicates that the proof is nearing Dafny's capabilities. In comparison we have also created the recursive proof in two different files. These files are *recursive_main_abstract* and *recursive_main_exact*. Since the recursive proof has been stabilized as much as possible the difference in verification time is minimal. The recursive abstract proof takes 2.6 while the recursive exact proof takes 2.7. Note that the proofs of the recursive files have the exact same amount of proof obligations while the proofs of the iterative files do not. These extra proof obligations in the exact iterative algorithm are a result of some additional assert statements to assist the Dafny verifier. These assert statements are not required in the abstract proof.

# Bibliography


[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005. 26

[2] Mike Barnett and K Rustan M Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87, 2005. 26

[3] Marcello M Bonsangue and Joost N Kok. The weakest precondition calculus: Recursion and duality. *Formal Aspects of Computing*, 6(1):788–800, 1994. 26

[4] Ran Chen, Cyril Cohen, Jean-Jacques Levy, Stephan Merz, and Laurent Théry. Formal Proofs of Tarjan's Strongly Connected Components Algorithm in Why3, Coq and Isabelle. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *ITP 2019 - 10th International Conference on Interactive Theorem Proving*, volume 141, pages 13:1 – 13:19, Portland, United States, September 2019. Schloss Dagstuhl–Leibniz-Zentrum fur Informatik. 13

[5] Ran Chen and Jean-Jacques Lévy. A semi-automatic proof of strong connectivity. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 49–65. Springer, 2017. 13

[6] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, USA, 1st edition, 1997. 26

[7] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. 4, 26

[8] Wesselink Huizing. Verifying tarjan's strongly connected components algorithm using dafny. Unpublished, June 2020. 1, 13

[9] Ioannis T Kassios. Dynamic frames and automated verification. In *2nd COST Action ICO701 Training School*. Citeseer, 2011. 28

[10] K Rustan M Leino. This is boogie 2. *KRML*, 178(131):9, 2008. 25

[11] K Rustan M Leino. Specification and verification of object-oriented software. *Engineering Methods and Tools for Software Safety and Security*, 22:231–266, 2009. 28

[12] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010. 28

[13] Rustan Leino. *Program Proofs*. unpublished, May 2020. Draft version. 26

[14] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972. iii, 1, 6

[15] Wieger Wesselink. State space exploration. Found on mcrl2.org, 2020. 6

[16] J. Öqvist. Iterative tarjan strongly connected components in python. https://llbit.se/?p=3379. Accessed: 2010-3-26. 1, 6

# Appendix A

# Implementation Dafny verification

All Dafny code verifying Tarjan's algorithm for SCC can be found in the GitHub directory:

https://github.com/WouterSchols/tarjanDafnyVerification

The main algorithms annotated with lemmas are contained in the files "*recursive_main_exact.dfy*" and "*iterative_main_exact.dfy*". The annotated proof of the recursive algorithm, algorithm 14, is contained in file "*recursive_main_exact.dfy*" and the annotated proof of the iterative algorithm, algorithm 14, is contained in file "*iterative_main_exact.dfy*". These proofs use lemmas which are based on the functions introduced in chapter 6. These functions describe the effect of sections of code in the Tarjan's algorithm. The algorithms were initially verified by substituting the sections of code by their respective functions. Using the functions directly improves readability and reduces load on the SMT solver which simplifies the implementation and debugging process. The annotated proofs where the code sections are substituted by functions are found in the files "*recursive_main_abstract.dfy*" and "*iterative_main_abstract.dfy*". We recommend looking at these files before looking at the exact files. The definitions of all functions, data types and invariants can be found in the files "*graph_defs.dfy*", "*recursive_defs.dfy*" and "*iterative_defs.dfy*". All other files contain the required lemmas.

The code is verified using pre-release 1 of Dafny version 3.0.0.20820 from 21-8-2020. The directory "*Dafny binaries*" contains the binaries of this Dafny version. The directory *Results* contains the Dafny output with is obtained by verifying all files with the trace, next to this this directory also contains some aggregated data from these runs. The last directory "*Dafny pre-existing*" contains the pre-existing Dafny verification introduced in chapter 4. This verification does not verify in Dafny version 3.0.0.20820.

# Appendix B

# Deterministically selecting items from a set

Sets and sequences are both collections commonly used in Dafny. A set is an unordered collection of elements where every element is unique. Sequences are stricter than sets in that they are an ordered collection of elements which can contains multiples. Since sets enforce less restrictions than sequences we prefer to use sets in proofs if possible. Because sets have less restrictions sets often enforce less proof obligations which can greatly simplify a proof. Sets do have one large disadvantage which makes them cumbersome in program code. This occurs because selecting an element from a set is usually not deterministic while program code has to be deterministic.

Consider the non empty set $xs$ and the Dafny code:

$$\textbf{var } x :| x \textbf{ in } xs$$

This code says let $x$ be a variables in set $xs$. This is not a deterministic operation which means that we can use this operation in proofs but we cannot use it in methods. This is why sequences are preferred in program code. Consider the set selection to the non empty sequence $ys$ and the Dafny code:

$$\textbf{var } y := ys[0]$$

Sequences allow us to easily deterministically select elements from a collection. This is why sets are preferred in proofs and sequences are preferred in methods. It can be convenient to convert a set to a sequence or a sequence to a set. This is why we define the function `SequenceToSet` and the method `SetToSequence`. We can easily write a function `SequenceToSet` which converts a sequence to a set, algorithm 15. This algorithm simply iterates over all elements in the sequence and adds them to a set recursively.

**Algorithm 15** SequenceToSet

**Input**: $ys$: A sequence.
`SequenceToSet`($ys$):
1: **if** $ys = [\,]$ **then**
2: **return** $\{\,\}$
3: **else**
4: **return** $\{ys[0]\} \cup$ `SequenceToSet`$(ys[1..])$

Creating a set to a sequence is more difficult. Note that if we can deterministically select an item $x$ from any non empty set $xs$ then we can use the same recursive pattern as the `SequenceToSet` function. Items can only be selected from a set using the **let** expression. A let expression $x :| P(x)$ adds the proof obligation that there exists at least one value of $x$ such that the predicate $P$ holds. A let expression is deterministic if we can prove that the value which satisfies $P$ is unique.

We want to select items from a set so let $P$ be $x \in xs \wedge Q(x)$. Now we need to pick $Q$ such that at least one element in $xs$ satisfies $Q$ given that $xs$ is not empty, $xs \neq \{\} \Rightarrow \exists x \in xs : Q(x)$. Next to this we need to prove that the element which satisfies $Q$ is unique, ie:

$$\forall x, x' \in xs : (Q(x) \wedge Q(x') \Rightarrow x = x')$$

A good candidate predicate for $Q$ is the predicate $\texttt{MAX}(x, xs)$ which states that $x$ is the larger or equal than all elements in the set $xs$.

$$\texttt{MAX}(x, xs) \equiv \forall x' \in xs : x' \leq x$$

The predicate $x' \leq x$ is usually predefined or easy to define. A proof which states that there exists a maximum and that the maximum is unique is trivial. This predicate has the convenient side effect that the resulting sequence is sorted. Using the `Max` preducate we can define method $\texttt{SetToSequence}(xs)$, algorithm 16.

**Algorithm 16** SetToSequence

**Input**: $xs$: A set.
$\texttt{SetToSequence}(xs)$:

```
1: if xs = { } then
2:     return [ ]
3: else
4:     var x  : | x ∈ xs ∧ Max(x, xs)
5:     return [x] + SetToSequence(xs \ x)
```

# Appendix C

# Lemmas recursive Tarjan's algorithm

In chapter 6 and chapter 7 lemmas are introduced to implement the verification of the recursive and iterative version of Tarjan's algorithm. In these chapters the lemmas are informally introduced but the exact definitions are not given. The lemmas have large and complex preconditions. The preconditions are required to verify the lemmas however the exact definitions do not add much to the overall proof. There are 11 different lemmas in the verification of `StrongConnect`.

## C.1 Lemmas DataValid

We want to ensure that the invariants in *data.Valid* are maintained. There are four locations in the Tarjan's algorithm where the global state is edited. A function was introduced in each of these sections. We want to prove that these functions maintain the *data.Valid* predicate. These four lemmas are *AddNewNodeDataValid*, *UpdateFromLowDataValid*, *UpdateFromDiscDataValid* and *PopFromStackDataValid*. These lemmas are defined in definitions C.1, C.3, C.3 and C.4. Each lemma ensures that the update section maintains the invariants. Furthermore, every lemma has some preconditions which can trivially be derived from the context information.

**Definition C.1** (AddNewNodeDataValid)**.** The lemma *AddNewNodeDataValid*(*data* : *TarjanData*, *u* : *Node*) ensures that the function *AddNewNode* maintains *data.Valid*. We require *PreconditionStrongConnect* to ensure that *u* is not yet in *stack* and the start state is valid.
*AddNewNodeDataValid*(*data*, *u*) ≡
**precondition** *PreconditionStrongConnect*(*data*, *u*)
**postcondition** *AddNewNode*(*data*, *u*).*Valid*()

**Definition C.2** (UpdateFromLowDataValid)**.** The lemma *UpdateFromLowDataValid*(*data* : *TarjanData*, *u* : *Node*, *v* : *Node*) ensures that the function *UpdateFromLow*(*data*, *u*, *v*) maintains *data.Valid*. This lemma requires a lot of context information. All this context information can trivially be derived from the postcondition of `StrongConnect(v)` since the postcondition ensures that *u* is not removed or edited in the global state.
*UpdateFromLowDataValid*(*data* : *TarjanData*, *u* : *Node*, *v* : *Node*) ≡
**precondition** *data.Valid*()
**precondition** $u \in data.stack$
**precondition** $u \in data.low$
**precondition** $v \in data.stack \Leftrightarrow data.disc[v] \neq data.low[v]$
**precondition** $v \in successors(u)$
**precondition** $data.disc[u] < data.disc[v]$
**postcondition** *UpdateFromLow*(*data*, *u*, *v*).*Valid*()

**Definition C.3** (UpdateFromDiscDataValid)**.** The lemma *UpdateFromDiscDataValid*(*data* : *TarjanData*, $u$ : *Node*, $v$ : *Node*) ensures that the function *UpdateFromDisc*(*data*, $u$, $v$) maintains *data.Valid*. This invariant requires that $u$ and $v$ are both in the stack.
*UpdateFromDiscDataValid*(*data*, $u$, $v$) $\equiv$
**precondition** *data.Valid*()
**precondition** $u \in$ *data.stack*
**precondition** $v \in$ *data.stack*
**precondition** $v \in$ *successors*($u$)
**postcondition** *UpdateFromDisc*(*data*, $u$, $v$)*.Valid*()

**Definition C.4** (PopFromStackDataValid)**.** The lemma *PopFromStackDataValid*(*data* : *TarjanData*, $i$ : $\mathbb{N}$) ensures that the function *PopFromStack*(*data*, $i$) maintains *data.Valid*. This lemma requires the invariant $S_9$(*data*, $i$). This invariant can be derived from the invariant group *LoopInvariant*.
*PopFromStackDataValid*(*data*, $u$, $v$) $\equiv$
**precondition** *data.stack*[$i$] $= u$
**precondition** *LoopInvariant*(*oldData*, *data*, $u$, |*successors*($u$)|)
**postcondition** *PopFromStack*(*data*, $u$, $v$)*.Valid*()

## C.2 Lemmas Loop invariant

The loop from line 5 to 10 in recursive `StrongConnect` algorithm, algorithm 2, maintains the invariant *LoopInvariant*. We require four lemmas to verify that the invariant *LoopInvariant*. One lemma is required to prove that the loop invariant holds at the start if the loop and one lemma is required for each path trough the loop. Note that the fact that the invariant group *data.Valid* holds in every state is used in every state of Tarjan's algorithm.

The first lemmas *AddNewNodeLoopInvariant* proves that the loop invariant holds at the start of the loop. This lemma is defined in definition C.5. The lemmas *RecursionLoopInvariant*, *UpdateFromDiscLoopInvariant* and *MaintainsLoopInvariant* are defined in definition C.6, C.7 and 6.8.

**Definition C.5** (AddNewNodeLoopInvariant)**.** The lemma *AddNewNodeDataValid*(*data* : *TarjanData*, $u$ : *Node*) ensures that the invariant group *LoopInvariant* after the *AddNewNode* section. This lemma requires the invariants *AddNewNode*(*data*, $u$)*.Valid* and *PreconditionStrongConnect* as precondition. The invariant *PreconditionStrongConnect* is required to ensure invariant $S_2$ is maintained after node $u$ is added to the *stack*.
*AddNewNodeDataValid*(*data*, $u$) $\equiv$
**precondition** *AddNewNode*(*data*, $u$)*.Valid*()
**precondition** *PreconditionStrongConnect*(*data*, $u$)
**postcondition** *AddNewNodeLoopInvariant*(*data*, $u$)

**Definition C.6** (RecursionLoopInvariant)**.** The lemma *RecursionLoopInvariant*(*oldData* : *TarjanData*, *preData* : *TarjanData*, *data* : *TarjanData*, $u$ : *Node*, $v$ : *Node*, $i$ : $\mathbb{N}$) ensures that the loop invariant holds after a recursive call. This is the most complex lemma and requires a lot of effort. This lemma requires two nodes and three states as input. The first state *oldData* should be the state at the start of a call. The second state *preData* should be the state at the start of the iteration. The third state *data* should be the state after the recursive call. The node $u$ should be the node which is being processed and $v$ should be the $i^{th}$ successor of $u$. All these states should be in a valid state and the invariant *PostconditionStrongConnect*(*recStart*, $v$, *data*) should hold. This lemma proves that the invariant *LoopInvariant* is maintained after the recursive call and the function *UpdateFromLow*.
*RecursionLoopInvariant*(*oldData*, *preData*, *data*, $u$, $v$, $i$) $\equiv$
**precondition** $v \notin$ *data.low*

**precondition** $oldData.Valid()$
**precondition** $preData.Valid()$
**precondition** $data.Valid()$
**precondition** $UpdateFromLow(data, u, v).Valid()$
**precondition** $PostconditionStrongConnect(preData, v, data)$
**precondition** $successor(u)[i] = v$
**precondition** $LoopInvariant(oldData, preData, u, i)$
**postcondition** $LoopInvariant(oldData, UpdateFromLow(data, u, v), u, i + 1)$

**Definition C.7** (UpdateFromDiscLoopInvariant)**.** The lemma $UpdateFromDiscLoopInvariant($ $oldData : TarjanData, data : TarjanData, u : Node, v : Node, i : \mathbb{N})$ ensures that the loop invariant is maintained in the branch $v \notin data.low \wedge v \in data.stack$. This lemma requires that the states $data$, $oldData$ and $UpdateFromDisc(data, u, v)$ are in a valid state.
$UpdateFromDiscLoopInvariant(oldData, data, u, v, i) \equiv$
**precondition** $v \notin data.low \wedge v \in data.stack$
**precondition** $oldData.Valid()$
**precondition** $data.Valid()$
**precondition** $UpdateFromDisc(data, u, v).Valid()$
**precondition** $successor(u)[i] = v$
**precondition** $LoopInvariant(oldData, data, u, i)$
**postcondition** $LoopInvariant(oldData, UpdateFromLow(data, u, v), u, i + 1)$

**Definition C.8** (MaintainsLoopInvariant)**.** The lemma $MaintainsLoopInvariant(oldData :$ $TarjanData, data : TarjanData, u : Node, v : Node, i : \mathbb{N})$ ensures that the loop invariant is maintained in the branch $v \notin data.low \wedge v \notin data.stack$. In this branch the state $data$ is not updated. This lemma requires that the states $data$ and $oldData$ are in a valid state.
$MaintainsLoopInvariant(oldData, data, u, v, i) \equiv$
**precondition** $v \notin data.low \wedge v \notin data.stack$
**precondition** $oldData.Valid()$
**precondition** $data.Valid()$
**precondition** $successor(u)[i] = v$
**precondition** $LoopInvariant(oldData, data, u, i)$
**postcondition** $LoopInvariant(oldData, UpdateFromLow(data, u, v), u, i + 1)$

## C.3 Lemmas deriving the post condition

We require two lemmas to verify the postcondition of `StrongConnect`. After the successor loop there are two cases if $disc[u] = low[u]$ and $disc[u] \neq low[u]$. In the first case $disc[u] = low[u]$ then we prove that the postcondition holds after the *PopFromStack* section. In the second case $disc[u] \neq low[u]$ then we can create one lemma which derives the postcondition holds using the loop invariant *LoopInvariant*.

The first lemma is *Proof1PostConditionStrongConnect* and this lemma is defined in definition C.9. This lemma states that the postcondition holds after $PopFromStack(data, |oldData, stack|)$. The second lemma is *Proof2PostConditionStrongConnect* which is defined in definition C.10. This lemma directly derives the postcondition from the loop invariant.

**Definition C.9** (Proof1PostConditionStrongConnect)**.** The lemma $Proof1PostconditionStrongConnect(oldData : TarjanData, data : TarjanData, u : Node)$ ensures that $PostconditionStrongConnect(oldData, PopFromStack(data, |oldData.stack|, u)$ holds. This lemma requires the *LoopInvariant* as precondition. The design of the *LoopInvariant* is designed to make the derivation of the postcondition trivial.
$Proof1PostconditionStrongConnect(oldData, data, u) \equiv$
**precondition** $oldData.Valid()$
**precondition** $data.Valid()$

**precondition** $PopFromStack(data, |oldData.stack|).Valid()$
**precondition** $data.low[u] = data.disc[u]$
**precondition** $LoopInvariant(oldData, data, u, v, |successors(u)|)$
**postcondition** $PostconditionStrongConnect(oldData, PopFromStack(data, |oldData.stack|), u)$

**Definition C.10** (Proof2PostConditionStrongConnect)**.** The lemma $Proof2PostconditionStrongConnect(oldData : TarjanData, data : TarjanData, u : Node)$ ensures that $PostconditionStrongConnect(oldData, data, u)$ holds. This lemma requires the *LoopInvariant* as precondition. The design of the *LoopInvariant* is designed to make the derivation of the post-condition trivial.
$Proof2PostconditionStrongConnect(oldData, data, u) \equiv$
**precondition** $oldData.Valid()$
**precondition** $data.Valid()$
**precondition** $data.low[u] \neq data.disc[u]$
**precondition** $LoopInvariant(oldData, data, u, v, |successors(u)|)$
**postcondition** $PostconditionStrongConnect(oldData, data, u)$

## C.4 Lemma PreconditionStrongConnectHolds

The lemma $PreconditionStrongConnectHolds(data, u, v)$ ensure that the precondition of `StrongConnect` holds before a recursive call. This lemma ensures that if $data.Valid$ holds and $u$ is connected to $v$ then $PreconditionStrongConnect(data, v)$ holds. This lemma is defined in definition C.11. The precondition $S_2(data)$ is part of the loop invariant.

**Definition C.11** (PreconditionStrongConnectHolds)**.** The lemma $PreconditionStrongConnectHolds(data : TarjanData, u : Node, v : Node)$ ensures that if $data.Valid$ holds and $u$ is connected to $v$ then $PreconditionStrongConnect(data, v)$ holds.
$PreconditionStrongConnectHolds(data, u, v) \equiv$
**precondition** $data.Valid()$
**precondition** $Connected(u, v)$
**precondition** $S_2(data)$
**postcondition** $PreconditionStrongConnect(data, v)$

This lemma *PreconditionStrongConnectHolds* can be difficult to prove. The problem is that we have to prove that the invariant $S_2(AddNewNode(data, v))$ holds. Since invariant $S_2(data)$ holds we can prove that $S_2(AddNewNode(data, v))$ holds by proving that all nodes in the stack are connected to $v$. Since $v$ is connected to $u$ we can prove that every node in the *stack* is connected to $u$. Let $w$ be a node in the stack. We want to prove $Connected(w, v)$. To do this we distinguish 2 cases.

1. Case $w \prec u \vee w = u$: By lemma $S_3$ we know that $disc[w] \leq disc[u]$ so by lemma $S_2(data)$ we can conclude $Connected(w, v)$.

2. Case $u \prec w$: By $S_1$ we know that $w$ is connected to a node $x$ such that $low[w] = disc[x]$. By $S_8$ we know that $low[w] \neq disc[w]$ so $x \prec w$. If $x \prec u \vee x = u$ then by case 1 we know $Connected(x, v)$ so we can conclude $Connected(w, v)$. Otherwise $u \prec x$ now once again $x$ is connected to some node below $x$. Eventually a node has to be reached which is in case 1. From case 1 we can conclude $Connected(w, v)$ so we conclude $Connected(w, v)$.

This concludes all cases and proves that invariant $S_2$ holds after the *AddNewNode* section.