Deductive Verification of Imperative Graph Algorithms

Gishor Sivanrupan

Supervisors: Arshavir Ter-Gabrielyan, Prof. Dr. Peter Müller
ETH Zürich

September 15, 2018

Contents

1. Introduction

1.1. Abstract

The goal of this project is to design a frontend extension of the Viper infrastructure, that allows to encode graph properties without the need to rewrite them into first order logic. For doing that, we used the known expertise in manually verifying graph algorithms, and tried to implement them in such a manner that this expertise is applied automatically.

We have realized that there is a lot of expertise for specific graphs like directed acyclic graphs. Because of that we have designed a type system, which the frontend uses for applying the appropriate wisdoms for the specific graphs.

Our frontend is able to verify several graph algorithms, and the user does not need to write the expertise by himself. Additionally, our frontend is able to check the type invariants of graphs after state updates. However, in bigger graph algorithms, the frontend does not always insert the needed wisdoms.

1.2. Viper

Viper is a verification infrastructure used for verifying properties of programs written in an intermediate verification language called the Viper language. It allows to reason in separation logic. However, it is not directly possible to specify higher order topological properties of graphs like reachability or acyclicity in the Viper language. Reachability can be simulated in first-order logic([3]), but it requires manually writing axioms. Manually writing them is error-prone and hence could lead to unsoundness.

1.3. Goal

We will design a language extension of Viper, and a frontend that translates the language extension to a valid Viper program. The frontend adds axioms automatically, such that it is possible to verify graph algorithms without losing the amout of automation Viper has. Afterwards, we will work with manually proven graph algorithms to test the performance of the frontend.

2. Manual Approach

In this chapter, we will describe what ingredients are needed to be able to verify graph properties like acyclicity in a manually written graph algorithm in the Viper language. For that, we will consider a small Viper method shown in Listing 2.5, which appends a node to the last element of the list.

2.1. Preliminaries

In this section, we will briefly describe some parts of the program, which are needed to write graph algorithms.

2.1.1. Nodes

We define objects on the heap as nodes. We can access them by having references to them. In Viper, we define `Ref` typed variables as node objects. Hence, whenever we talk about node typed variables, we actually mean `Ref` typed variables.

2.1.2. The `GRAPH` Macro

The `GRAPH` macro takes a set of references `nodes` as an input. It expresses that `null` is not in the set and that we have access to all node typed fields of the elements of the set.

For every node typed field `$field$`, we add require that

```
forall n:Ref :: {n.$field$} n in nodes ==> acc(n.$field$))
```

holds. If we had only one node field called `next`, we would define a graph as in Listing 2.1.

Closedness

Additionally, we can constrain that all node typed fields are also contained in `nodes` or store the value of `null`. For a field `next`, we would describe it as in Listing 2.2.

```
define GRAPH(nodes)
    !(null in nodes)
    && ( forall n:Ref :: {n.next} n in nodes ==> acc(n.next) )
```

Listing 2.1.: The `GRAPH` macro

```
( forall n:Ref ::
        {n.next in nodes}
        {n in nodes, n.next}
            n in nodes && n.next != null ==> n.next in nodes )
```

Listing 2.2.: The closedness property for the field `next`

2.1.3. The `Edge` Domain

`Edge` is an abstract domain type, which contains several functions and axioms. For example, it contains the `create_edge(p:Ref, s:Ref): Edge` to create directed edges between two nodes.

The complete `Edge` domain is given in Appendix A.0.1

2.1.4. The `TrClo` Domain

The `TrClo` domain encodes the `edge` relation and its transitive closure, the `exists_path` relation. Both functions take as arguments an edge set `EG` and two nodes `start` and `end` and return `Bool`.

The first one, `edge`, returns `true` if and only if `EG` contains `create_edge(start, end)`

The second one, the `exists_path` function is used to get the transitive closure of the `edge` function.

Furthermore, it contains the functions `acyclic_graph(EG: Set[Edge])`,

`func_graph(EG: Set[Edge])`, and `unshared_graph(EG: Set[Edge])`, and their corresponding axioms.

For `acyclic_graph`, the axiom specifies that for any two nodes `v1` and `v2`, there cannot be an edge from `v1` to `v2`, and at the same time a path from `v2` to `v1`.

For `func_graph`, it says that if one node `v` has an edge to the node `v1` and `v2` in `EG`, then `v1 == v2` must hold

For `unshared_graph`, it says it analogously in the other direction, e.g if two nodes `v1` and `v2` have an edge to `v`, then `v1 == v2` must hold.

```
function $$(refs:Set[Ref]): Set[Edge]
    requires forall n:Ref :: {n.next} n in refs ==> acc(n.next)
    ensures forall p:Ref, s:Ref ::
        { create_edge(p,s) }
            p in refs && s in refs && p.next == s
            <==> create_edge(p,s) in result
```

Listing 2.3.: The `$$` function

The complete `TrClo` domain is contained in Appendix A.0.2

2.1.5. The $$ Function

The `$$(g: Set[Ref])` function returns the set of edges of the graph represented by g. More precisely, the result of the function is the set that contains exactly the elements satisfying the following conditions:

1. they are representable by `create_edge(p, s)`, for some nodes `p` and `s`.
2. `p` and `s` are such that `p in g` and `s in g`, and `p` has a node typed field referenced to `c`

The function is shown in Listing 2.3.

2.1.6. `apply_TCFraming` Function

The `apply_TCFraming(g0: Set[Ref], g1: Set[Ref])` function is used when we need transitive closure related information about the union of two graphs. The problem is that `exists_path` of `g0` relations are not automatically added to `g0 union g2` by the frontend, as it is unclear, how to preserve the negative `exists_path` relation between two nodes.

The `apply_TCFraming` function generally does not give us the complete transitive closure relation, but approximates it.

For any two nodes for which there exists a path in one of the two graphs, we can be sure that there is also a path in the union of the two graphs. For any two nodes, for which there does not exist a path in one of the two graphs, we can also conclude that there does not exist a path in the union of them.

The `apply_TCFraming` function is defined in Appendix A.0.4

```
method link(g:Set[Ref], x:Ref, y:Ref)
    requires x in g
    requires y != null ==> y in g
    requires PROTECTED_GRAPH(g,x)
    requires x.next == null
    ensures PROTECTED_GRAPH(g,x)
    ensures x.next == y
    ensures y == null ==> $$(g) == old($$(g))
    ensures y != null ==> forall v1:Ref, v2:Ref ::
        { edge(old($$(g)),v1,v2) }
            edge($$(g),v1,v2) <==> edge(old($$(g)),v1,v2) ||
               (v1==x && v2==y)
    ensures y != null ==> (forall v1:Ref, v2:Ref ::
        { exists_path($$(g),v1,v2) }
            (v1 != v2) ==> (
                exists_path($$(g),v1,v2) <==>
                    exists_path(old($$(g)),v1,v2) ||
                    ↪  (exists_path(old($$(g)),v1,x) &&
                    ↪  exists_path(old($$(g)),y,v2))))
```

Listing 2.4.: The list link method

2.1.7. The Link and Unlink Methods

Precise update formulas for the existing path relation are not known for destroying edges in general graphs. However, there are precise update formulas for specific graphs.

For acyclic list segments, we use the link and unlink, with update formulas defined in [3]. The Link method is defined in Listing 2.4, where `PROTECTED_GRAPH(g, x)` means that we only have full access to the node fields with `x`. For all other nodes in `g`, we only have read permissions.

We additionally have link and unlink methods for directed acyclic graphs (DAGs).

Furthermore, we have link and unlink methods for Zero or one path graphs. These are graphs, if for each pair of nodes `u` and `v`, there is at most one path of length ≥ 1 from `u` to `v`. That means for example that cycles are allowed, however, nested cycles are not allowed.

We have the update formulas for both types of graphs from [1]. The link and unlink methods are specified in the Appendix.

```
field next: Ref
method append(g:Set[Ref], x:Ref) returns (new_g:Set[Ref], y:Ref, last:
    ↪ Ref)
{
    var old_g:Set[Edge] := $$(g)
    last := x
    while ( last.next != null )
    {
        last := last.next
    }
    y := new(*)
    y.next := null
    new_g := g union Set(y)
    assume apply_TCFraming(g, Set(y))
    link(new_g,last,y)
    assume trigger(forall u:Ref, v:Ref :: !exists_path($$(new_g),u,v) ==>
    ↪  u != v && forall w:Ref :: !edge($$(new_g),u,w) ||
    ↪  !exists_path($$(new_g),w,v))
    assume trigger(forall u:Ref, v:Ref :: !exists_path($$(new_g),u,v) ==>
    ↪  !edge($$(new_g),u,v)        )
}
```

Listing 2.5.: The append method

2.2. The append Method

This method should create a new node and add it to an acyclic list segment.

2.2.1. Parameters

The method has an input argument `g`, which is a set of references, and a reference `x`. More concretely, g is an acyclic list segment that starts with the node `x`. The output arguments are `new_g`, `y`, and `last`. `y` and `last` are references to nodes in the graph. `new_g` is a set of references, which again is an acyclic list segment.

2.2.2. Method Body

The method works as follows: First, it iterates over the list segment starting with `x`, until it reaches the last element of it, and assigns `last` to that element. Then, it assigns `y` to a fresh reference, initializes its `next` field with null, and assigns `new_g` to the union of the input graph `g` and the new node `y`. Finally, it links `last` with the new node y.

```
    requires GRAPH(g)
    requires x in g

    requires forall n:Ref :: n in g ==> exists_path($$(g),x,n)


    requires acyclic_graph($$(g))
    requires func_graph($$(g)) //trivial, if we only have one reference typed field.
    requires unshared_graph($$(g))


    ensures new_g == g union Set(y)
    ensures GRAPH(new_g)
    ensures acyclic_graph($$(new_g))
    ensures func_graph($$(new_g))
    ensures unshared_graph($$(new_g))
    ensures forall n:Ref :: n in new_g ==> ((exists_path($$(new_g), x, n) && (exists_path(old($$(g)), x, n) || n == y)))
    ensures forall n1: Ref, n2: Ref :: n1 in new_g && n2 in new_g ==> (edge($$(new_g), n1, n2) <==> (edge(old($$(g)), n1, n2) || (n1 == last && n2 == y)))
```

Listing 2.6.: The method specification of `append`

In the method body on line 14, we call the function `apply_TCFraming(g, Set(y))` and assume its result, because we need the `exists_path` relation of `g union Set(y)`.

The two assumptions on the lines 17 and 18 of Listing 2.5 are needed for triggering and have no other effect.

2.2.3. Method Specification

The footprint of the `append` is specified in Listing 2.6. In the prestate, we want that `g` is a closed graph, hence we apply the macro defined in Section 2.1.2. In the postcondition, we want that `new_g`, which is defined as `g union Set(y)` to be a closed graph.

Additionally, we want `x` to be an element of `g`. In the precondition, we want that for all nodes `n` in `g`, there exists a path from `x` to `n`, such that `x` is the starting node of the acyclic list segment `g`.

We want that in the precondition `g` is acyclic, functional, and unshared and in the postcondition, `new_g` should have these properties. As we only have one reference typed

```
        invariant GRAPH(g)
        invariant last in g

        invariant forall n:Ref :: n in g ==> exists_path($$(g),x,n)

        invariant acyclic_graph($$(g))
        invariant unshared_graph($$(g))

        invariant $$(g) == old_g
```

Listing 2.7.: The loop invariant

field in this example, graphs are always functional.

Furthermore, we want to check that the `exists_path` relation has been updated correctly. First of all, we want to check that for all references `n` in `new_g`, there exists a path from `x` to `n`. Secondly, for each node `n in new_g`, either there has been a path from `x` to `n` in the old graph, or `n` is equal to `y`. With these assertions, we ensure that only one new edge has been added.

2.2.4. The Loop Invariant

The loop invariant is specified in Listing 2.7. As we only iterate over the graph `g` without changing it, most conditions in the invariant are preserving the specifications from the prestate. The only additional invariant that we need for the loop counter is `last` remains in g.

2.3. Automation

In Section 2.1, we have shown several functions and domains, that need to exist and that we need to use before we can encode and verify graph algorithms in Viper. It should be possible to insert these parts of code automatically.

For example, the user needs to write several preconditions, to define a set of references as a graph. Then, one would need to write them again in the while loop invariant. Another example is that the user generally specifies graphs as a set of references. But to be able to reason about reachability properties, one would need to reason about the set of edges from the graph.

3. Frontend Language

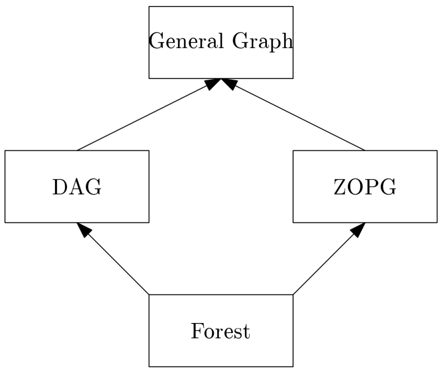
In this example, we will define the core elements of our frontend language. First, we will describe our graph and node type system, then we will specify functions and operations of the frontend.

3.1. Graph Type System

3.1.1. Motivation

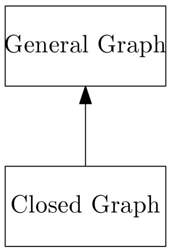
As mentioned earlier in Section 2.1.7, path update formulas after destroying edges in general graphs are not known. Hence, generally we cannot preserve the transitive closure after destructive updates. However, there are update formulas for for some classes of graphs: DAGs, ZOPGs, and Lists. While it is possible to specify acyclicity and "functionality" of graphs, it is fundamentally impossible to specify that a graph is a ZOPG in first-order logic.. However, we know how to check field updates and some binary set expressions such that they preserve the zero or one path property. Because of that, we have decided to create a type system, such that the user can specify some input graphs as ZOPGs. We take this as given, and we check at each state update that the ZOPG property is preserved. With that, it would be sound to use the update formulas.

DAGs and ZOPGs are not comparable: There are DAGs, which are not ZOPGs, and there are ZOPGs, which are not DAGs. For this reason, we call graphs, that are DAGs and ZOPGs, Forests. We get the following topology lattice:



The edges mean that the cell with the outgoing edge is a subtype of the cell with the incoming edge.

Additionally, we also want to differentiate between closed and general graphs. We will see later in section 5.1.3 that binary set expressions with closed graphs could result in a more concrete Topology than general graphs. We get the following closedness lattice:



In our frontend language, for every graph declaration, we would need to specify their static topology and closedness. We specify the concrete type as follows:

`Graph["Topology", "Closedness"]` In place of "Topology", we can write the following types:

- `Forest`
- `DAG`
- `ZOPG`
- `_`, which means that the graph has a general topology.

In place of "Closedness", one can write two types:

- `Closed`
- `_` which means that it is unknown, if the graph is closed or not.

The type of any graph begins with the Keyword `Graph`. If the type attributes are emitted, then this means that it is a general graph. Generally, one can only use the type system in the method parameters and in the local variable declarations in the method body. One cannot use them in functions, domains. Also, one cannot use them in forall expressions, even if it is in a method.

3.2. Nodes

If we declare a variable of type node, we need to specify in the type parameters, to which graph they belong. A node type is specified as follows:

`Node["Graph"]`

There are again several options of what to write in place of "Graphs":

- One can write just one graph, for example `Node[g]`. This means that the node is either in the specified graph or null.

- It is possible to write multiple graphs (`Node[g0, g1]`) This means that the node is either in the union of the specified graphs, or null.
- One can write `Node[_]`, which means that the graph is in the universe. That means for input nodes that it either is in the union of the input graphs or null. For non null output nodes, it means that it must be in the union of the input and output graphs. For non null nodes in the method body, it means that it is in one of the existing graphs, or it is a fresh node.

One can again leave the type attributes and only write `Node`. This is equivalent to `Node[_]` in methods. Again, it is not possible to write type arguments in `forall` expressions and anything except method parameters and local variable declarations inside the the method body.

3.3. Universe Variable

We have added the variable `UNIVERSE` to all methods. It is initialized as the union of all input graphs. Whenever a local or output graph is assigned, or a fresh node is created, we add the new nodes to the `UNIVERSE`.

3.4. Graph Literals

We have also added the possibility to write `Graph()` in order to specify explicit graphs. It is handled equivalently as explicit Sets with elements of type `Ref`. That means that between the brackets, one can specify the nodes, which should be element of the explicit graph.

3.5. Functions

We have added the following functions to the specification of the frontend:

1. `EDGE` This function takes one graph argument, and two nodes, and is translated to the `edge` function. Additionally, one could also only put the two nodes as arguments. Then, the frontend will add `UNIVERSE` or the union of input (and output) graphs as graph argument.
2. `EXISTS_PATH` This function takes the same arguments as `EDGE`, and is translated to the `exists_path` function. One could again only pass two node arguments. It would behave the same way as `EDGE`.
3. `DISJOINT` This function takes two input graph parameters. It checks, if the two graphs do not intersect.

4. `ACYCLIC` takes one graph parameter `g`, and is translated to `acyclic_graph($$(g))`. Hence, it specifies that the graph is acyclic.
5. `FUNCTIONAL` This function also takes one graph parameter `g`, and is translated to `func_graph($$(g))`. That means that every node can only have an outgoing edge to at most one node.
6. `UNSHARED` This function applies the function `unshared_graph`. That means that every node can only have an incoming edge from at most one node.
7. `CLOSED` This function specifies that the passed graph is closed.
8. `IS_INITIALIZED` This function is used for local and output graph variables. This may be useful for the user, if a local graph `g` has been initialized inside of a loop, such that the frontend is not able to register that the graph is initialized after the while statement. In that case, the frontend is not able to automatically insert the loop invariants of subsequent while statements, such that it is preserved that `g` is initialized. The user can manually write in the loop invariants that `IS_INITIALIZED(g)`. Assuming this function could lead to unsoundness.

3.6. Operations

We have added the following operations to the frontend:

1. `UPDATE` This operation is a generalization of a field update `x.next := y`. We need this generalization, as a field update updates the set of edges twice: First, when the field gets unlinked, and then when it gets linked. This operation takes four or three arguments: The first one is the name of the field. The second one is the graph, on which the update has to be executed. This argument is optional, the frontend would put `UNIVERSE`, if it is left out. The last two arguments specify the two nodes that should be connected. The first node will be connected to the second.
2. `UNLINK` This operation is similar to the previous one, except that it only unlinks.
3. `LINK` This is the analogue link operation.
4. `NEW` This operation is used to create a fresh node. It is assured that all node typed fields are initialized to `null`.

The operations `UPDATE_ZOPG`, `UNLINK_ZOPG`, `LINK_ZOPG`, `UPDATE_DAG`, `UNLINK_DAG` and `LINK_DAG` are defined similarly to `UPDATE`, `UNLINK`, and `LINK` except that they use the corresponding update operations.

3.7. Fields

Fields can be of type `Node`, but they cannot take any type arguments. They cannot be of type `Graph`.

There are several magic fields, that set the settings of the frontend. They get deleted for verification:

1. `__CONFIG_OUROBOROS_INLINE` This field enables function inlining. That means that functions defined by the frontend are getting inline.
2. `__CONFIG_OUROBOROS_TYPE_CHECK` This field adds type checking. Without this field, the frontend believes, that the user knows which update operationss are sound to use.
3. `__CONFIG_OUROBOROS_UPDATE_INVARIANTS` This field adds update invariants before link methods. It behaves differently, depending on if type check is on or off. If type check is off, it will check the update invariants of the graph, which has been specified in the link method, depending on the specified topology in the link method. Otherwise, it will check all update invariants of all graphs depending on whether they are affected by this update or not.

4. Translation

In this chapter, we will discuss how the frontend will translate the program written in the frontend language, such that we get a valid Viper program, and that the user may be able to verify graph algorithms.

4.1. Preamble

In each verification of a program, the frontend will add all the operations from section 2, and synthesize them correctly, such that the field names of the given program are taken, and the operations are correct adapted to the number of node fields.

Because of that, the user is able to apply functions and methods from the preamble, without specifying them.

4.1.1. Method Specifications

If a method has any input or output graphs, the frontend will insert the corresponding pre- and postconditions automatically for each graph. Particularly, it will add that `null` is not contained in the union of input graphs. Additionally, it will give us the access permissions for `Node` typed fields of the elements of the graphs. We need these permissions in order to specify graph properties like closedness and acyclicity. Lastly, it will add for each input Graph `g` the topology and closedness properties. That means that for each closed graph, it will require that all `Node` fields from the elements in the graph are either `null` or are also contained in the graph. For each DAG, it requires acyclicity. For ZOPGs, it will requires that a function `IS_ZOPG` holds. This function is not specified, as ZOPGs cannot be specified in first-order logic.

In the postcondition, it will ensure that `null` is not contained in the union of input and output graphs, and that we have the access permissions for `Node` typed fields. The properties of each graph are expressed in the same manner as in the precondition.

The declared types of the graphs specify type invariants that have to hold in the prestate and in the poststate. If one wants that some input graph has some property in the prestate of the method, one would need to write them in the precondition. For example, if one wants an input graph `g` of type Closed Forest, but an output graph of

```
field left: Node
field right: Node
method specs(
    g0: Graph[ZOPG, _], g1: Graph, x: Node[g0]) returns (
    g2: Graph[DAG, Closed], y: Node)
requires DISJOINT(g0, g1)
requires CLOSED(g0)
```

Listing 4.1.: The frontend specifications of the method `specs`.

a ZOPG, one needs to declare the graph as `g: Graph[ZOPG, _]`, and add in the precondition the two function applications `ACYCLIC(g)` and `CLOSED(g)`. In that way, the static type of `g` is declared as `Graph[ZOPG, _]`, but the dynamic type in the prestate is `Graph[Forest, Closed]`

For every input node in the method formal arguments, the frontend inserts the precondition that its value is either `null` or that it is contained in the declared graph. For every output node, the frontend inserts the postcondition analogously. We have added this constraint, because with this restriction, we know that this node is either null, or we have the access permissions to `Node` fields. Additionally, if we want to do a field update, we need a graph, on which we can perform this update. In that way, we are know at least that each Node is contained in the `UNIVERSE`.

As an example on how the methods' pre- and postconditions are inserted, we will consider the method specification in the Listing 4.1.

The Viper encoding of the method `specs` is shown in the Listing 4.2. The requirement that `null` is not contained in the union of the input graphs, is added on line 4. The frontend requires the access permissions for all `Node` typed fields from the graphs on line 5 to 8. `g0` is a ZOPG, and `g1` is a general graph. Hence, we do only add the precondition `IS_ZOPG(g0)`. The specification of node `x` is added on line 9. The next two lines are the user-specified preconditions in the frontend language.

In the postconditions, the frontend again inserts the null check and the access permissions first. Afterwards, the frontend adds the closedness and acyclicity checks of `g2`. Finally, the node specification is also checked.

4.1.2. Local Variable Declarations

In this section, we will describe how the frontend will handle local graph and node variables in the method body. Additionally, we will show how the `UNIVERSE` variable works.

```
method specs(
       g0: Set[Ref], g1: Set[Ref], x: Ref) returns (
       g2: Set[Ref], y: Ref)
  requires !(null in g0 union g1)
  requires (forall n: Ref :: { n.left } (n in g0 union g1) ==>
              acc(n.left, write))
  requires (forall n: Ref :: { n.right } (n in g0 union g1) ==>
              acc(n.right, write))
  requires x != null ==> (x in g0)
  requires DISJOINT(g0, g1)
  requires CLOSED(g0)
  ensures !(null in g2 union g0 union g1)
  ensures (forall n: Ref :: { n.left } (n in g2 union g0 union g1) ==>
             acc(n.left, write))
  ensures (forall n: Ref :: { n.right } (n in g2 union g0 union g1) ==>
             acc(n.right, write))
  ensures CLOSED(g2)
  ensures acyclic_graph($$(g2))
  ensures y != null ==> (y in g0 union g1 union g2)
```

Listing 4.2.: The Viper encoding of the method `specs`

Graph Variable Declarations

If a graph variable is declared, the frontend creates a unique variable. We need a unique variable which would specify if the graph has been initialized other not. If the graph has not been initialized, we cannot get the access permissions for the input graphs. Without the permissions, we are also not able to check the type of the graph. Hence, we treat uninitialized graphs as if they do not exist.

The type of the variable depends on if type checking is enabled. If that is the case, it creates a `Int` variable, and initializes it with `0`. Otherwise, it creates a `Bool` variable, and assigns it to `false`.

The reason for the two different type is because when we type check, we will do a reaching definitions analysis. Then, after a graph assignment, we would assign the corresponding `Int` variable to a unique value, such that we know that if the variable has that unique value, then this concrete assignment reaches.

As an example, if we disable type checking, and declare a graph as:

```
var g: Graph[_, _]
```

, then the frontend translates this declaration to:

```
var g: Set[Ref]
var g_init: Bool := false
```

Node Variable Declarations

If a node variable is declared, the frontend checks to which corresponding graph it has assigned to by the user, and inhales that the node is either `null` or in that graph.

For example, if we declare a node `var n: Node[g0, g1]` for some graphs `g0` and `g1`, the frontend adds the statement after the declaration:

```
inhale n != null ==> n in g0 union g1
```

4.1.3. Graph Assignments

If a graph `g`, which has a unique variable `g_init`, is assigned, the frontend sets `g_init` such that it specifies that `g` has been initialized.

4.1.4. Fresh Variable Assignment

There are two possibilities, how a node can be assigned to a fresh reference: We can use the Viper `new()` method, or the frontend `NEW()` method.

If we use the `NEW()` method call, the frontend will declare a new node variable, and assign it to the result of the method call `create_node(UNIVERSE)`, which returns a node outside of the `UNIVERSE` with access permissions to all reference typed fields. Additionally, it expands the `UNIVERSE` by this new node variable. Finally, it initialized the node, which the user initially wanted to assign to new node, to the new node variable.

The reason, why we use this approach is that now the frontend internally has a variable which points to the fresh node. This variable can be used in the loop invariant, to preserve that the `UNIVERSE` contains the fresh node. Additionally, the `create_node` method ensures that all Node typed fields are initialized with `null`.

The `create_node` method is a bodyless method and is shown in Listing 4.3.

If we use the `new()` method, we can specify to which fields we want to have access. The frontend can only handle cases, when we have access to all reference typed fields. Otherwise, it will throw an error. If all reference typed fields are specified, the frontend will call the `NEW()` method to get a fresh node with access permissions to all node fields. Additionally, it will inhale that we have access to all other specified fields in the `NEW()` method.

Consider the example in Listing 4.4. The translated version of the fresh node assignment is shown in Listing 4.5. The translated version only contains the assignment. As `val` is a

```
method create_node(u: Set[Ref]) returns (node: Ref)
  requires !((null in u)) && (forall n: Ref :: { n.next } (n in u) ==>
                                        acc(n.next, 1 / 2))
  ensures !((null in u)) && (forall n: Ref :: { n.next } (n in u) ==>
                                     acc(n.next, 1 / 2))
  ensures !((node in u))
  ensures acc(node.next, write) && node.next == null
```

Listing 4.3.: The `create_node` method

```
field val: Int
field next: Node
method create()
{
  val x: Node
  x := new(*)
}
```

Listing 4.4.: Viper program with a method, that creates a fresh node

`Int` typed field, `create_node` does not provide access permission to that field. Hence, it is inhaled on line 6. Line 5 is a node check, that x remains in the `UNIVERSE`. As `UNIVERSE` has been updated one line before, this assertion is trivially true. We still do this check, because it could have been that the user has declared the node to be part of a smaller graph. Then, we would need to add an error.

4.1.5. If-then-Else and While Statements

The frontend translates *if* or *while* statements, when fresh nodes are created in their bodies. If that is the case, it creates a new variable `var body_scope: Bool`, initializes it on top of the body to `false`, and inside the statement body to `true`. It collects the

```
1   var fresh_x: Ref
2   fresh_x := create_node(UNIVERSE)
3   var x: Ref
4   x := fresh_x
5   UNIVERSE := UNIVERSE union Set(x)
6   assert x in UNIVERSE
7   inhale acc(x.val, write)
```

Listing 4.5.: Translation of the body of create from Listing 4.4

variable to be set to true, to support the fresh node generation. It uses that for generating the loop invariants.

4.1.6. Loop invariants

The frontend uses the `UNIVERSE` variable to preserve that `null` is not in any of the existing graphs and that it has access permissions to all reference typed fields. Additionally, it preserves that all existing graphs are subsets of the `UNIVERSE`.

The frontend adds an invariant to preserve that we have access to all Node fields in the universe.

In the loop invariants, the frontend needs to preserve some important properties of each existing graph.

We define existing graphs as follows: All input graphs are existing graphs. All local and output graphs are existing graphs, if they have been initialized. Otherwise, they are handled as empty graphs.

In order to check this dynamically, we express a local graph `g` depending on its variable `g_init`, that specifies, if `g` is initialized. More concretely, we express `g` as `cond ? g : Set[Ref]()`, where `cond` is either the expression `g_init` or `g_init != 0`, and it is `true`, if `g` is initialized.

If the frontend knows that the graph has been initialized, it will insert the invariant `g_init` or `g_init != 0`, respectively. However, sometimes it is not possible for the compiler. Hence, the user has the possibility to add the invariant `IS_INITIALIZED(g)`, to preserve that the graph is initialized.

For all nodes, that have been constructed by the NEW method, we want that the fresh node is also contained in the universe, independently of whether the user still has a local variable initialized to the fresh node. Recall that the frontend has internally created new variables for fresh nodes in Section 4.1.4. Additionally, we have collected, which scope variable (from Section 4.1.5) has to be true, such that the internally created variable is assigned to a fresh node. We use this information to specify that if the scope values is true, then the fresh node is contained in the `UNIVERSE`.

Finally, the frontend preserves that the local and output nodes are in the graphs that have been specified in the declaration. The frontend preserves the same expression, that it inhaled, when the node has been declared.

5. Type Checking

In this chapter, we will first discuss the type rules of graphs and nodes. Afterwards, we will discuss, when we would need to check the types, and how. Types are checked by taking the static types, and if they are not enough, assertions are added to the code.

5.1. Graph Type Rules

We can formulate closedness and acyclicity in separation logic in the Viper language. Hence, if we want to check if some expression has some specific graph type, we may want to first check if it is of the the non-closed and cyclic version of that type, and then add the closed and acyclic proof obligations, if needed.

In this section, we will show how we can check the type of variables, explicit sets, binary set expressions, conditional set expressions, and method calls.

5.1.1. Set Variables

If we want to check, if a graph variable g fulfils a specific type, we first check if the declared type of g is a subtype of the specified type. In that case, we do not need to add any additional assertions.

Otherwise, we need to check the type of the variable. We do it in the following way: We collect all reaching definitions of g, and check that the type is fulfilled in each of the reaching definitions.

5.1.2. Graph Literals (Explicit Sets)

For all explicit sets, we first need to ensure that `null` is not in the set. Afterwards, we check that we have all access permissions to all node typed fields.

Generally, we check if the explicit set is of some specific type by applying the functions `ACYCLIC`, `CLOSED` and `IS_ZOPG`. However, there are some special cases:

Empty Set: An empty set can be of any type. Hence, we never need to add any assertions for type checking empty sets.

Singleton Set: A singleton set can be of any non-closed type. If we want to check closedness, we simply check it with the closedness function.

In listing 5.1 on line 12, we have the following line:

```
var newGraph: Graph[Forest, Closed] := Graph(newNode)
```

After this line, we would need to check if we have access permissions for all node typed fields of `newNode`. Additionally, as `Graph(newNode)` is a singleton graph, we know that it is automatically of type Forest, but we need to check closedness. Hence, we need to add the following assertion:

```
assert !(null in Set(newNode)) && (
  (forall n: Ref ::
    { n.next }
    (n in Set(newNode)) ==>
    acc(n.next, write)) && CLOSED(Set(newNode)))
```

5.1.3. Binary Set Expressions

We can support three binary set operations, which result in a set: union, intersection and setminus. We also differentiate whether the two operands are disjoint or not. For each combination, we have created a table, where the header row and the header column describe the type of the two operand graphs, and the cells describe the type of the resulting graph. The header row describes the type of the right hand side, and the header column the type of the left hand side.

Set Union

The union operation is commutative. Hence, the union tables are symmetric.

Disjoint Set Union The resulting graph cannot be declared to have a more concrete type than a general graph if both operands are non closed.

For example, if one of the operands is already typed as a general graph, the resulting graph, will also be typed as a general graph. Additionally, if we want to have a union of a DAG and a ZOPG, the resulting graph might be a general graph. This comes from the fact that DAGs and ZOPGs are not comparable. Even if we take the union of two ZOPGs, we only get a Graph, because it is possible that there are several edges from one ZOPG to the other, such that there are also several paths to some nodes in the other ZOPGs.

The same holds when we take the union of two DAGs, because it could be that one node from one DAG has an edge to another node from the other DAG, and vice-versa.

However, if we know that one graph is closed, then we may be able to say more about the type of the resulting graph. For example, if both Graphs are DAGs, and one of them is closed, we can conclude that the resulting Graph is also a (non closed) DAG. This follows from the fact that only paths from the non closed DAG to the closed DAG are possible. But there cannot be paths in the other direction, which are needed to get cycles.

For ZOPGs, we will not get more concrete types, as edges from one graph to the other are sufficient to violate the zero or one path property.

If both graphs are closed, we are first of all able to say that the resulting Graph is also closed. Additionally, we get a more concrete type for two closed ZOPGs: As both graphs are disjoint, and do not have any outgoing edges, we can conclude that the union of them is also a closed ZOPG.

disjoint union	Graph	Closed Graph	DAG	Closed DAG	ZOPG	Closed ZOPG	Forest	Closed Forest
Graph	Graph	Graph	Graph	Graph	Graph	Graph	Graph	Graph
Closed Graph		Closed Graph	Graph	Closed Graph	Graph	Closed Graph	Graph	Closed Graph
DAG			Graph	DAG	Graph	Graph	Graph	DAG
Closed DAG				Closed DAG	Graph	Closed Graph	DAG	Closed DAG
ZOPG					Graph	Graph	Graph	Graph
Closed ZOPG						Closed ZOPG	Graph	Closed ZOPG
Forest							Graph	DAG
Closed Forest								Closed Forest

General Set Union After creating the table, we have realized that the general set union gives the same types as the disjoint set union. For the cases where the resulting graph is a general graph, this follows trivially, as the general set union also contains the cases of the disjoint set union.

When both graphs are DAGs, and one of them is also closed, we again get a DAG as a result. This follows from the fact that if the two graphs intersect with each other, it must hold that the connected segment of the non closed DAG, that intersects with the other graph, must be a subset of the closed DAG. The reason is that if it was a superset of the closed DAG, that would mean that the closed DAG cannot be closed, which is a contradiction. We can remove the intersecting segment of the non closed DAG, such that we get a disjoint set union.

If both graphs are closed ZOPGs and intersect with each other, we know that there are no edges from the intersecting part to the other parts of the graphs. Additionally, we know that the edges from the other parts to the intersecting part do not violate the ZOPG property. Hence, we get a special case of a disjoint set union of the closed intersecting part with the non closed other parts, where we know that the edges from the other parts to the intersecting part are constructed in such a way, that the ZOPG property holds.

union	Graph	Closed Graph	DAG	Closed DAG	ZOPG	Closed ZOPG	Forest	Closed Forest
Graph	Graph	Graph	Graph	Graph	Graph	Graph	Graph	Graph
Closed Graph		Closed Graph	Graph	Closed Graph	Graph	Closed Graph	Graph	Closed Graph
DAG			Graph	DAG	Graph	Graph	Graph	DAG
Closed DAG				Closed DAG	Graph	Closed Graph	DAG	Closed DAG
ZOPG					Graph	Graph	Graph	Graph
Closed ZOPG						Closed ZOPG	Graph	Closed ZOPG
Forest							Graph	DAG
Closed Forest								Closed Forest

Set Minus

In the set minus, we always know that we have at least the topology of the left hand side, because we cannot violate some DAG or ZOPG property by removing nodes. However, the closedness property could be lost.

Disjoint Set Minus If the two operands are disjoint, the result of this operation will always be the left hand side. Hence, it is sufficient to take the type of the left hand side.

disjoint setminus	Graph	Closed Graph	DAG	Closed DAG	ZOPG	Closed ZOPG	Forest
Graph	Graph	Graph	Graph	Graph	Graph	Graph	Graph
Closed Graph	Closed Graph	Closed Graph	Closed Graph	Closed Graph	Closed Graph	Closed Graph	Closed Graph
DAG	DAG	DAG	DAG	DAG	DAG	DAG	DAG
Closed DAG	Closed DAG	Closed DAG	Closed DAG	Closed DAG	Closed DAG	Closed DAG	Closed DAG
ZOPG	ZOPG	ZOPG	ZOPG	ZOPG	ZOPG	ZOPG	ZOPG
Closed ZOPG	Closed ZOPG	Closed ZOPG	Closed ZOPG	Closed ZOPG	Closed ZOPG	Closed ZOPG	Closed ZOPG
Forest	Forest	Forest	Forest	Forest	Forest	Forest	Forest
Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest

General Set Minus In the general case, the topology of the result of a setminus will always be at least the topology of the left hand side, as we cannot make an acyclic graph cyclic or add new paths by removing nodes. However, it could be that the closedness property gets lost, as in a graph, we could remove one node which is reachable by some other node.

setminus	Graph	Closed Graph	DAG	Closed DAG	ZOPG	Closed ZOPG	Forest	Closed Forest
Graph	Graph	Graph	Graph	Graph	Graph	Graph	Graph	Graph
Closed Graph	Graph	Graph	Graph	Graph	Graph	Graph	Graph	Graph
DAG	DAG	DAG	DAG	DAG	DAG	DAG	DAG	DAG
Closed DAG	DAG	DAG	DAG	DAG	DAG	DAG	DAG	DAG
ZOPG	ZOPG	ZOPG	ZOPG	ZOPG	ZOPG	ZOPG	ZOPG	ZOPG
Closed ZOPG	ZOPG	ZOPG	ZOPG	ZOPG	ZOPG	ZOPG	ZOPG	ZOPG
Forest	Forest	Forest	Forest	Forest	Forest	Forest	Forest	Forest
Closed Forest	Forest	Forest	Forest	Forest	Forest	Forest	Forest	Forest

Intersection

Disjoint Intersection The intersection of two disjoint graphs is always an empty set. As an empty is always a closed forest, we will always get a closed forest as result.

disjoint intersection	Graph	Closed Graph	DAG	Closed DAG	ZOPG	Closed ZOPG	Forest
Graph	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest
Closed Graph	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest
DAG	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest
Closed DAG	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest
ZOPG	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest
Closed ZOPG	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest
Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest
Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest	Closed Forest

General Intersection The general intersection is similar to the general setminus. The difference is, that in the general setminus, the topology of the left hand side remains, while in the disjoint set minus the topology of both operands remain.

intersection	Graph	Closed Graph	DAG	Closed DAG	ZOPG	Closed ZOPG	Forest	Closed Forest
Graph	Graph	Graph	DAG	DAG	ZOPG	ZOPG	Forest	Forest
Closed Graph	Graph	Graph	DAG	DAG	ZOPG	ZOPG	Forest	Forest
DAG	DAG	DAG	DAG	DAG	Forest	Forest	Forest	Forest
Closed DAG	DAG	DAG	DAG	DAG	Forest	Forest	Forest	Forest
ZOPG	ZOPG	ZOPG	Forest	Forest	ZOPG	ZOPG	Forest	Forest
Closed ZOPG	ZOPG	ZOPG	Forest	Forest	ZOPG	ZOPG	Forest	Forest
Forest	Forest	Forest	Forest	Forest	Forest	Forest	Forest	Forest
Closed Forest	Forest	Forest	Forest	Forest	Forest	Forest	Forest	Forest

5.1.4. Conditional Set Expressions

Conditional set expressions are ternary expressions, consisting of a Boolean condition, the *then* and *else* operands. If the condition holds, we take the then operand, otherwise, we take the else operand. In conditional set expressions, we check if the then operand and the else operand have the wished type. If the then operand is not of wished type, we say that the conditional expressions is not of the wished type, if the condition does not hold. We do it analogously for the else operand.

5.2. Node Type Rules

We can check the types of a node by asserting that the node is contained in its declared graph.

5.3. Generating Proof Obligations

5.3.1. Field update

A Field update consists of a unlink and a link method. The unlink method may never violate any graph type invariants, as it only removes edges.

Before the link method, we might need to check what is linked. Assume we want to link x with y. Then, for every closed graph of any topology, that contains x, we need to check that y is also contained in the graph. For every DAG, that contains x and y, we need to check that there is no path from y to x, if y and x are different nodes. Otherwise, we would get a cycle. The DAG update would be encoded as follows:

```
!(exists_path($$(g), y, x))
```

For every ZOPG, that contains x and y, we need to check that for all two non equal nodes u and v, that already have path from u to v, there should not be also a path from u to x and from y to v. Otherwise, we would have created a new path from u to v.

```
(forall u: Ref, v:Ref ::
  {exists_path($$(g),u,x), exists_path($$(g), y, v)}
  u in g && v in g && u != v && exists_path($$(g), u, v) ==>
       !(exists_path($$(g), u, x)) ||
       !(exists_path($$(g), y, v)))
```

For all Forests, we need to check the DAG and ZOPG invariants.

In addition to the checks, the information about the nodes can be used to improve the update methods. For example, if we want to link `x` with `y`, e.g. `x.next := y`, we normally update the `exists_path` relation of `UNIVERSE`. But the `UNIVERSE` is a general graph. That means that we will use the general unlink method, which does not contain any update formulas for `exists_path`. If we knew that `x` and `y` are contained in the same graph `g`, which is of type DAG or ZOPG, we can use the precise update formulas for those topologies to get the updated `exists_path` relation on `g`. In combination with framing, we could get the update formulas for the `UNIVERSE`.

5.3.2. Method Call

In case of a method call, we need to check several things before and after the call.

Before the call, we need to check for every call argument of type graph that the passed argument has the type of the formal argument. We do not have to check that the node arguments are in the declared graphs of the formal arguments, as this is already done in the preconditions of the method.

After the call, we need to make sure that the types of the formal returns of the call graph targets are a subtype of the declared type of the targets. Additionally, we need to check for every other graph that intersects with the call graph argument, that they still have the declared type. Furthermore, we have to guarantee that every node target is contained in its declared graph. Lastly, we have to assert that every node, that has been declared to potentially be an element of any of the graph targets, is still contained in the declared graph.

5.3.3. Graph Assignment

After a graph assignment, we first need to check that the right hand side of the assignment has the declared type of the left hand side. This can be done by the type rules defined in section 5.1.

Secondly, we need to assure that the nodes, that have been declared to be a potential element of the assigned graph, are still elements of their declared graph.

5.3.4. Node assignment

After a node assignment, we need to add an assertion, that ensures that the nodes remained in the declared graph.

5.4. Example

In this section we will look at a slightly changed version of the append method in Listing 2.5. In this version, we do not have a list, but a Forest. Additionally, we already get the node `x`, which has to be linked with the new node.

5.4.1. Node check

On line 14 of listing 5.1, we have assign `x_union`, which is defined to be in `unionGraph`, to `x`. Afterwards, we need to check that `x_union` is in `unionGraph` or is `null`. That means that we add the following assertion:

```
assert x_union != null ==> (x_union in unionGraph)
```

After line 15, we would add an analogous assertion for `n_union`

5.4.2. Graph check

In the example , there are several things, that need to be checked by the frontend.

```
field __CONFIG_OUROBOROS_TYPE_CHECK: Bool
field __CONFIG_OUROBOROS_UPDATE_INVARIANTS: Bool
field next: Ref
method main(g: Graph[ZOPG, _], x: Node[g]) returns (unionGraph:
 ↪  Graph[Forest, Closed])
requires x != null && x.next == null
requires ACYCLIC_LIST_SEGMENT(g)
requires CLOSED(g)
requires ACYCLIC(g)
{
    var newNode: Node[_]
    newNode := new(*)
    var newGraph: Graph[Forest, Closed]
    if(newNode != null) {
        newGraph := Graph(newNode)
    }
    unionGraph := g union newGraph
    var x_union: Node[unionGraph] := x
    var newNode0: Node[unionGraph] := newNode
    x_union.next := newNode0
}
```

Listing 5.1.: The frontend method `append_Forest`

Assignments to Graph Literals

First of all, on line 14, `newGraph` is assigned to an explicit graph. After this assignment, the frontend needs to check that we have access permissions to all `Node` typed fields from the nodes in `Graph(newNode)`. Additionally, we would need to check that the explicit graph is a closed forest. Because it is a singleton graph, it is trivially a forest. Closedness needs to be checked.

Reaching Definitions and Set Expressions

For each local and output graph, we create a local int variable, which we initialize with zero. Whenever the local graph is assigned, we assign the local int with a unique value. For example, in the Listing 5.1, we add after line 14

```
newGraph_init := 1
```

On line 16, we want to ensure that `g union newGraph` is a closed forest. There are several possibilities, to check that a union of two graphs is a closed forest. By looking at our table, we decide to check these four possible types of the two graphs:

1. `g` and `newGraph` are closed forests.
2. `g` and `newGraph` are closed ZOPGs. Additionally, we need to check acyclicity of `g union newGraph`.
3. `g` is a closed ZOPG and `newGraph` is a closed forest. Additionally, we need to check acyclicity of `g union newGraph`.
4. `g` is a closed forest and `newGraph` is a closed ZOPG. Additionally, we need to check acyclicity of `g union newGraph`.

We know that `newGraph` is typed as a closed forest. Additionally, `g` is a ZOPG. Hence, we can add the following assertion:

```
assert CLOSED(g) && acyclic_graph($$(g))
```

However, because of the `if` statement at line 13, the frontend has two reaching definitions of `newGraph` at line 16. One of the reaching definitions has the unique value 0, which means that `newGraph` has not been initialized. Hence, the frontend adds another assertion:

```
assert newGraph_init != 0
```

In fact, the front end adds a disjunction of assertions for each of the cases. But in this case, it is sufficient to add the two mentioned statements.

Field update

At line 19, there is a field assignment. Generally, we would translate this to

```
unlink_next(UNIVERSE, x_union)
link_next(UNIVERSE, x_union, newNode0)
```

Because of the definitions of `x_union` and `newNode0`, we know that both variables are either `null` or are elements of `unionGraph`. Combined with the fact that `unionGraph` is a Forest, we can use the DAG update formulas:
`link_DAG_next(unionGraph, x_union)` and
`link_DAG_next(unionGraph, x_union, newNode0)`. However, we still need to check the update invariants of all affected graphs. We need to check the following statements

1. `newNode` is in `unionGraph`.
2. There is no path from `newNode` to `x_union` in `unionGraph`.
3. If both nodes are in `g`, then we need to check the update invariant of the ZOPG `g`.
4. If both nodes are in `newGraph`, then we need to check the update invariant of the ZOPG, and DAG `newGraph`.

By looking at the code, one could realize that all of these assertions already hold. The frontend would add assertions for every case.

6. Implementation

We have implemented the described frontend from the previous chapters. In this chapter, we will discuss, how we have implemented it.

The frontend is a plugin of Viper. A plugin of Viper has the possibility to change the structure of the Code at several stages of the verification. We mostly changed the structures of the Parse Abstract Syntax Tree (Parse AST) and the Abstract Syntax Tree (AST), to translate the code to a valid Viper program.

6.1. Phases

The process of translating frontend code to Viper code can be split into two major phases, which again consist of smaller phases.

We have designed frontend language such that it is parseable by Viper. Hence, the frontend does not need to do anything until the parse tree has been created b< Viper.

6.1.1. Parse Tree

In the first phase, we transform the Parse AST, such that the subsequent type check of Viper does succeed. For that, we will need to add missing operations, transform the types to Viper types, and some more things. In this chapter, we will describe what has to be done.

User Options

First of all, we will collect the magic fields defined in the section 3.7 of the program, such that the optional phases are enabled, if needed. After having collected the user options, the frontend deletes the magic fields.

Names Handling

Afterwards, we collect all definition names, like function names or local variable declaration names, and check if any of them is equal to one of the reserved keywords defined in chapter 3. If that is the case, then we add an error to Viper, such that the verification process terminates with a failure message. Otherwise, we also collect all `Ref` or `Node` typed fields.

Function Synthesizing

We load the preamble of predefined domains, operations, functions and wisdoms.

Field update operations are specified for each node field separately, hence, we need to copy the operation for each node field. The preamble uses the (not existing) field `$field$`, which we will replace by the node fields of the frontend program and copy the resulting operation, such that we get one for each node field.

Wisdoms and functions are not specified for each node separately, but may use all node fields at once. We then handle it depending on the function. For example, the `CLOSED` function, has a precondition of the form:

```
(forall n: Ref :: { n.$field$ } (n in g) ==> acc(n.$field$, 1 / 2))
```

The frontend copies this precondition, replaces `$field$` with with the name of each node field name and returns the conjunction of them as the new precondition.

Finally, the frontend renames the definition names, if some identifiers already occured in the names collection from the previous step. It saves the name changes for later purposes.

Method Handing

Afterwards, the frontend collects for each method the input graphs and nodes and the output graphs and nodes with their types.

The frontend will add several things to the method body, if one exists:

It adds a `UNIVERSE` variable on top of the method body. Afterwards, it will iterate over the body, and add to any local variable declaration into a function application, such that the type informations of the local variable declarations are preserved. For example, if we have a following graph declaration

```
var g: Graph[DAG, Closed] ,
```

we will translate it to the following statements:

```
var g: Set[Ref]
inhale CLOSED_DAG_decl(g)
```

Analogously, `var n: Node[g0, g1]` will be translated to:

```
var n: Ref
inhale Node_decl(n, g0 union g1)
```

For all output graphs and nodes, it will also inhale a function application on top of the body.

We check for all `new` statements that it contains all node fields. If that is not the case, we report an error at this stage.

We will translate all `Graph` typed variables to `Set[Ref]` typed variables, and all `Node` typed variables to `Ref` typed variables, because otherwise, we cannot use set expressions like `in` between `Node` variables and `Graph`. Hence, type information of variables gets lost after this step and we need to collect these informations here.

Wisdoms and Operations Translation

Then, the frontend will translate some wisdoms applications defined in section 3.5 like `EXISTS_PATH` to the corresponding Viper function calls to functions like `exists_path` from the frontend. For example, `EXISTS_PATH` will be translated to `exists_path` and if the first argument was the graph `g`, then this will be changed to the set of edges `$$(g)`.

Furthermore, we provide the wisdom `PROTECTED_GRAPH` that is mainly used for the preamble, but the user can also apply them. It has two different signatures:

```
PROTECTED_GRAPH(g: Set[Ref])
```

or

```
PROTECTED_GRAPH(g: Set[Ref], node: Ref, fieldName: String)
```

In the first case, the wisdom is handled in sch a way that we have `1 / 2` access permissions to all fields.

In the second case, the wisdom is handled in such a way that we have `1 / 2` access permissions to all fields, except for the specified `node`, which has full write permission for the field with the name `fieldName`.

Analogously, the field update operations are being translated. The signature of the `UPDATE` method is:

```
UPDATE(fieldName: String, g: Set[Ref], from: Ref, to: Ref)
```

or

```
UPDATE(fieldName: String, from: Ref, to: Ref)
```

In the first case, the operation is translated to the update method for the given specific name `fieldName`. In the second case, the graph `g` is additionally set automatically to `UNIVERSE`.

6.1.2. Abstract Syntax Tree

After having translated the Parse AST such that type checking does not fail, the frontend receives the AST. It will iterate multiple times over the method bodies in the AST to translate it to the right Viper program.

Most of the steps, that are done in this phase could have also be done in the Parse AST. However, in the Parse AST, error transformation is not possible. Hence, only if we translate the method in the AST, we are able to report our own errors.

Method specifications

The frontend uses the collected input graphs and nodes and the output graphs and nodes with their types to add the correct specifications of the method described in section 4.1.1.

Additionally, the frontend inhales two wisdoms in the beginning of the method body, if the body exists:

It adds the Framing wisdom described in section 2.1.6 for any two input graphs.

Furthermore, it adds the `apply_no_exit_closed` wisdom for any two input graphs `g0` and `g1`, where `g0` needs to be a closed graph. This wisdom is similar to the No Exit Axiom described in [3], and adds the fact that when `g0` and `g1` are disjoint, then there cannot be a path from any node in `g0` to any node in `g1`, as `g0` is closed.

First Iteration

In the first iteration over the method body, the frontend will handle local variables, the `UNIVERSE` variable, and the loop invariants.

While it iterates over the method body, it will maintain and update the following lists:

- A list of local Nodes and their declared type arguments.
- A list of local Graphs, its type arguments, its unique variable to check if it is initialized, and a graph state, which says if the graph is always initialized.
- A scope variable declaration, which defines if we are in a While or If scope.

- A list of newly created node variables by the frontend, and in which scope they have been created.
- A list of newly created unique variables for checking if local graphs have been initialized.

For that, it maintains a list of local Nodes and their declared type arguments, a list of

In this subsection, we will discuss what the frontend will do for each type of a statement.

- **Method Body Block:** The frontend iterates over all statements inside the method body, and then initializes all scope variables with `false`, and all unique variables of the local graphs to either `false` or `0`, depending on whether type checking is enabled or not. It adds all the new variable declarations, that have been done during type checking, on top of the method body, such that it is reachable from anywhere.
- **While Statement:** The frontend adds several invariants to the statement. Generally, it uses the available lists to add the invariants described in the section 4.1.6. One optimization is done to the local graphs. In Section 4.1.6, a local graph `g` with its unique variable `g_init` is defined as `cond ? g : Set[Ref]()`, where `cond` is `g_init`, if type checking is disabled, and `g_init != 0` otherwise. In our implementation, if we know that `g` is always initialized on this line, we define the existing graph as `g`.

  Before iterating over the body of the statement, the frontend creates a new `Bool` variable for the while block scope. Then, it iterates over the block, takes the returned new block, and checks if any new node variables have been declared inside the block.

  If that is not the case, then it removes the newly created scope variable. Otherwise, it adds on top of the returned block an assignment of the scope variable to `true`, and before the while statement, it assigns the scope variable to `false`.
- **Inhale Statement:** Recall that we have inserted graph and node declaration inhales in the Parse AST. With the help of them, we can find out, if a graph or node has been declared with the inhale statement. If it is not a graph or node declaration, the inhale remains unchanged.
- **Graph Declaration Inhale:** If a graph has been declared, the frontend learns from its type by the name of the function application, and creates a unique variable for the local graph. It assigns the unique variable to either `false` or `0`. It saves the local graph with its unique variable and the fact that it is not initialized in its list of local graphs.

  If type checking is disabled, the frontend replaces the inhale by the unique variable assignment. Otherwise, it adds the unique variable to the arguments of the function application, such that it gets the name of the unique variable in the type checking phase.

For example, consider the inhale from section 6.1.1:

```
inhale CLOSED_DAG_decl(g)
```

If the unique variable has the name `g_init` and type checking is disabled, then it will replace the statement by

```
g_init := false
```

If type checking is enabled, it will replace the statement by

```
inhale CLOSED_DAG_decl(g, g_init)
g_init := false
```

- **Node Declaration Inhale:** If a node has been declared, the frontend learns from the graphs, in which it is contained, if its value is not `null`. We inhale that the node is either in the graph or its value is `null`. We again keep the inhale, if type checking is enabled. Otherwise, we remove it.

- **New Statement:** In a `new()` statement, the user can specify, which for fields one wants to have access permissions. We know that the user wants to have access to at least all the node fields, as otherwise we would have reported an error in the previous phase. But it is possible that the user also wanted access permissions to non node fields. We call the method `NEW()` to get a node with access to node fields, and the node fields are initialized with `null`. Then, we handle the `NEW` Method Call. How this is handled is described below.

  Additionally, we inhale that we have access to all the specified non node fields.

- **Field Assignment:** A field assignment will be translated to the corresponding update method.

  For example, if we want to assign the node typed `next` Field of a node `n` to `y`, we replace the field assignment with `update_next(UNIVERSE, n, y`, given that the user has not defined any entity by `update_next`. The field update method call is described below.

- **Local Variable Assignment:** Whenever a graph has been assigned, we use the available lists to check that the nodes, that have been declared to be potentially part of that graph, are either still contained in their declared graphs, or are `null`.

  Finally, we add in the list of collected graphs that the graph is now initialized. Then, if type checking is disabled, we assign its unique variable to `true`.

  Whenever a node has been assigned, the frontend add the check that either the node is still contained in its declared graph, or that its value is `null`.

- **`NEW` Method Call:** If `NEW` has been called, the frontend replaces the statement as described in Section 4.1.4. Additionally, it adds the newly created fresh variable to the list of fresh variables, and map it to the scope variable, if this statement has been called in a local scope.

- **Field Update Method Call:** The frontend operates as described in section **??**. It translates the update methods to separate unlink and link methods.
- **If Statement:** For the if-then and else blocks, The frontend creates a scope variable for each. It then iterates over each block, and checks whether it has created a new variable in the block. If that is the case, it sets the scope variable on top of the corresponding to `true`, and before the `If` statement to `false`. Otherwise, it removes the scope variable from the list of newly declared variables.

  Then, it checks for all local variables outside of the scope of the if-then and else block, if they have been initialized in both blocks. If that is the case, it adds to the list that the local variable is initialized, after the if statement.

Second Iteration

The second iteration is only executed, if type checking is enabled. Then, we iterate over each method body, do a Reaching Definitions Analysis, and check the types of the graphs. The types of the nodes have already been checked in the previous iteration. For this iteration, the frontend maintains a wrapper containing of maps, lists and a Boolean flag, namely

- A map of all input graphs and their types.
- A map of all input Nodes, and their graphs.
- A map of statements, that have already been copied for type checking, with a mapping from the original variables to the copied ones.
- A map of local graphs with their types, unique variables, and the highest value, to which the unique variable has been assigned to.
- A map of graphs with their reaching definitions. For each reaching definition, we additionally save the value of unique variable, and the wrapper at that state.
- A list of errors, that should be reported.
- A Boolean flag, stating if type checking should be executed or not.

With these lists, the frontend executes the reaching definitions analysis and type check.

Reaching Definitions Analysis In this paragraph, we will briefly describe, what the frontend will do in the reaching definitions analysis for specific statements. Generally, when the Boolean flag for type checking is set to false, the frontend does not change the statements. It only updates its lists.

```
//statement s
//state after_s
//state before_while
while(b)
{
 ...
}
//state after_while
```

Listing 6.1.: The states before and after the while block

- **Graph Declaration Inhale Statement:** We use the left inhale statement from the first iteration, to obtain the name, type, and the unique variable of the graph. We add it into the list of local graphs and set the highest value of the unique value to `0`.

  Additionally, we put in the list of reaching definitions, a fake definition to the declared graph, with the integer value `0`. We use this to check, whether the graph might be uninitialized.

  We remove the inhale statement, when the Boolean flag for type checking is set to true.

- **While Statement:** Consider the code in Listing 6.1. To obtain all reaching definitions at `before_while`, we need the union of the reaching defintions at `after_s` and at `after_while`. Because of that, we first iterate over the while block, without type checking, such that we get the reaching definitions from the while block to `after_while`. Then we join the these reaching definitions with the reaching definitions at `after_s`, to get the reaching definitions at `before_while`, under the condition that the definitions at `after_s` are complete. This is not always the case. Especially, when we are in a nested loop, it is possible, that we still do not have all the reaching definitions at the state `before_while`.

  Because of that, we iterate over the while block again, but only type check, if the Boolean flag is set to true. After the second iteration, we again take the union of the reaching definitions at `after_s` and `after_while` to get the resulting wrapper after the while block.

  Additionally, we compare the two maps from local graphs to the highest values of the unique variables, and take the mutual highest values for the resulting wrapper.

  We could probably optimize the second iteration over the while loop, such that it is only done when the original Boolean flag is set to true. However, the main performance issue is the time it takes to verify graph algorithms, and not the time it takes to translate the code. Furthermore, we do not get unsound by this extra iteration, as we do not change anything to the code, and we should get the same

```
1  method type_check(g0: Graph[_, _], g1: Graph, x: Node[g0], y: Node[g0])
2  requires x!= null && y != null && x != y
3  requires ACYCLIC(g0)
4  {
5      var g2: Graph := g0
6      var g3: Graph := g1
7      var g4: Graph := g2 setminus g3
8      g3 := g0
9      g2 := g1
10     UPDATE_DAG(next, g4, x, y)
11
12 }
```

Listing 6.2.: Type Check example

result after taking the union of the state before the while loop and the state after the second iteration.

- **If Statement:** In case we have an if statement, the frontend first copies the wrapper twice. Then, it iterates over the then-block with the first copy. Afterwards, it takes the second copy combined with the highest values of unique variables from the first copy, such that the unique variables are assigned to unique values in the else block. With that copy, it iterates over the else block.

  To get the resulting wrapper, it joins the reaching definitions of the two copies, and takes the highest values from the second copy.

- **Method Call And Graph Assignments** If a graph has been assigned by a method call or assignment, the frontend adds the method call to the reaching definitions of the graph, and deletes all other definitions for this graph.

Type Check How the types are checked, is described in the chapter 5.

One problem of checking types of the right hand side of a graph assignment, is that the value of the right hand side might have changed, if we evaluate it in the current state. Consider the example in the Listing 6.2. On line 4, `g4` is assigned to `g2 setminus g3`. `g2` at that state has been assigned to `g0`, which is a DAG.

But if we checked the type of `g2` at the state of line 9, we would have erroneously come to the conclusion that `g2` is assigned to `g1`, which is a general graph.

Hence, we do not only need the reaching definitions, but also the states of the definitions. For that, the frontend does always also collect the wrapper of the state of the reaching definition.

```
1  method test()
2  {
3     var g2: Graph
4     g2 := create_ACYCLIC()
5     var g3: Graph := g2
6     g2 := create()
7
8     var g4: Graph[DAG, _] := g3
9  }
10
11 method create_ACYCLIC() returns (g: Graph)
12 ensures ACYCLIC(g)
13 method create() returns (g: Graph)
```

Listing 6.3.: Dynamic Type Check example

Additionally, we also collect statement copies. The reason, why we use state copies, is because when we cannot prove some type statically, and need to check the type of the graph dynamically, we need the graph of the specific state.

For example consider the code in Listing 6.3. `g3` is assigned to the graph `g2` at line 5. `g2` has been assigned to the result of `create_ACYCLIC`, which returns a graph which is statically declared as a general graph, but with the post condition, we learn that it is actually a DAG. Hence, the assignment on line 7 is valid. The frontend would need to check that `g2` is acyclic at line 5. However, if he added the assertion `assert ACYCLIC(g)` after line 7, the program would fail. The reason is, because `g2` has been assigned to a different graph at line 6.

Hence, if we want to check the type of some definition, the frontend creates a state copy. That means that it creates two new variables for each local graph appearing on the right hand side: One to copy the graph, and one to copy its unique variable.

In this example, the frontend would have created `var g2_copy: Set[Ref]` and `var g2_init_copy: Int`, and would have assigned it to:

```
g2_copy := g2
g2_init_copy := g2_init
```

before line 5, where `g2_init` is the unique variable of `g2`. With that, the frontend can add the assertion that `assert ACYCLIC(g2_copy)` after line 7, and would succeed.

- **Unlink or Link Method Call** When the Boolean flag is set to true, the frontend will try to optimize the unlink or link method. If the graph, on which the update method is being executed, is the `UNIVERSE`, the frontend tries to optimize the graph, as described in section 5.3.1.

Afterwards, it tries to optimize the update formula has been specified as the general graph. Again, it optimizes it, as described in section 5.3.1.

Only for link methods, it will also add the update invariant described in section 5.3.1

- **Method Call** The frontend uses the wrapper, to check the types as described in Section 5.3.2.

  Additionally, the frontend adds for every graph, that has been assigned by this call, a new unique value. More concretely, it takes the highest value of the unique value saved in the mapping with the local graphs, and increments this value by one, and adds the assignment of this unique value to the code.

- **Graph Assignments** The frontend checks the types as described in Section 5.3.3. Furthermore, it adds an assignment of the unique variable to a unique value, as described in the case of method calls.

Unused Functions

In the last step of graph translation, the frontend collects all functions and method, that the translated program uses from the preamble. Every other function and method gets removed by the frontend. This is done, because if we had several node typed fields, many methods would have been added, which slows the verification down.

Inlining

If the user enables inlining, the frontend inlines the functions and the methods from the preamble.

Inlining Function Applications The frontend inlines all preamble functions, that contain a body. By doing that, the precondition gets removed. The preconditions of the functions are expressing the access permissions for the function parameters. That means that the access permissions will not be checked anymore. First of all, when the frontend has inserted the code by itself, then it is mostly already asserted, that the precondition holds. Hence, we remove a redundant check.

And if the precondition did not hold, the program will still fail, but with a different error message, as the functions need the access permissions described in the precondition for the body.

Inlining Inhale Function Applications If function applications do not have a body, and are inhaled, the frontend will inline them. For that, the frontend exhales all pure preconditions of the function. That means that statements about access permissions are not exhaled. After the exhale, the frontend inhales all the postconditions of the function.

Inlining Method Calls The frontend inlines the bodyless update methods. For that, it adds a label before the method call. Then, it exhales all preconditions and inhales all postconditions. But before inhaling the postconditions, it replaces all `old` expressions by labelled `old` expressions, such that the label immediately before the method call is used. Without doing that, it would be unsound to inline, because `old` would refer to the state before the method body has been executed.

7. Evaluation

In this chapter, we will handle some test cases, to evaluate, how our frontend performs compared to manually written code, and how much automation we achieve. Particularly, we will discuss, which wisdoms we could add automatically, such that in the frontend, one needs to manually apply as few wisdoms as possible.

7.1. Test Cases

We have chosen several examples of manually written graph algorithms from [4], and tried to verify them in the frontend, namely

- a list reverse algorithm
- a program that appends two list segments in one graph.
- an algorithm that joins two acyclic lists into one acyclic list and one that joins two DAGs into one DAG
- an algorithm that splits an acyclic list into two acyclic lists and one that splits a DAG into two DAGs
- an algorithm that inserts a new node into a cyclic list (a ring).

7.2. Results

Except the list reverse algorithm, which did not terminate in the frontend and in the manually written version, and the ring insert algorithm, we were able to verify all the selected programs. However, we needed to add some wisdoms to the programs such that they suceeded.

```
field next: Node
method unjoin(g0:Graph[DAG, _], g1:Graph[DAG, _],
              x0:Node[g0], x1:Node[g1])
    requires DISJOINT(g0,g1)
    requires CLOSED(g0 union g1)
    requires x0 != null && x1 != null
    requires forall n:Node ::
      { EXISTS_PATH(g0,n,x0) }
      n in g0 ==> EXISTS_PATH(g0,n,x0)
    requires forall n:Node ::
      { EXISTS_PATH(g1,x1,n), n in g1 }
      n in g1 ==> EXISTS_PATH(g1,x1,n)
    requires forall n:Node, m:Node ::
      { n in g0, m in g1 }
      n in g0 && m in g1 ==>
        (EDGE(g0 union g1,n,m) <==> n==x0 && m==x1)
    requires forall n:Node, m:Node ::
      { n in g1, m in g0 }
      n in g1 && m in g0 ==> !EDGE(g0 union g1,n,m)
    requires ACYCLIC_LIST_SEGMENT(g0 union g1)
    requires x0.next == x1
    ensures CLOSED(g0)
    ensures CLOSED(g1)
    ensures forall n:Node ::
      { EXISTS_PATH(g1,x1,n), n in g1 }
      n in g1 ==> EXISTS_PATH(g1,x1,n)
    ensures forall n:Node ::
      { EXISTS_PATH(g0,n,x0) }
      n in g0 ==> EXISTS_PATH(g0,n,x0)
{
    var unionGraph: Graph[DAG, Closed] := g0 union g1
    UNLINK(next, unionGraph, x0)
}
```

Listing 7.1.: The `unjoin` method for lists

7.2.1. Split Lists

Consider the example, that splits an acyclic list into two acyclic lists in Listing 7.1. This method takes two disjoint acyclic lists `g0` and `g1` with two nodes `x0` and `x1`, such that `x0` is the last element of `g0` and `x1` is the first element of the list `g1`. The edge from `x0` to `x1` is the only connection between the two graphs in the prestate.

We had the following postconditions:

1. `g0` and `g1` are closed graphs.
2. `g0` and `g1` are acyclic.
3. `x0` remains the end of the acyclic list `g0`.
4. `x1` remains the start of the acyclic list `g1`.

However, we were only able to prove the first condition.

The second postcondition could not be proved, even though we were able to show that `ACYCLIC(g0 union g1)` holds in the poststate. Hence, this was a framing problem. Adding `FRAMING(g0, g1)` in the end of the list solved the problem.

The last problem got also solved by adding the framing wisdom in the end.

The third postcondition, however, did still not verify, even though it did in the manual approach. The main difference between the manual approach and the translated code by the frontend was, that in the manual approach, the list update formula was used, while the frontend uses the DAG update formula. The DAG update formula for `EXISTS_PATH` is precise, also for acyclic lists.

Hence, the problem may be that the triggers of the postconditions of the `UNLINK_DAG` method are set unfavourable. Triggers are used to instantiate `forall` quantifiers. Actually, there are also two `exists` quantifiers in the postcondition. One could negate the whole quantifier, and the body of the quantifier to get a `forall` quantifier, such that one can find better triggers. We did not find triggers, which solved our problem.

We could also think about adding the topology List to our Type System.

7.2.2. Join Lists

Consider the example, that joins two acyclic lists in Listing 7.2. This method's structure is slightly different to the `unjoin` method in Listing 7.1: Instead of receiving a Node, on which we should update the field, we receive the start of both acyclic lists. Then, we first need to traverse over one acyclic list to get its last element, on which we need to apply the field update.

The method takes two acyclic and disjoint lists `g0` and `g1`, and their starting nodes `x0` and `x1`. Both lists are closed in the prestate. The method should return the nodes `x` and

```
field next:Ref
method join(g0:Graph[DAG,_], g1:Graph[DAG,Closed], x0:Node[g0], x1:Node[g1])
  returns (x:Node[g0], last:Node[g0])
    requires CLOSED(g0)
    requires x0 != null
    requires x1 != null
    requires ACYCLIC_LIST_SEGMENT(g0)
    requires ACYCLIC_LIST_SEGMENT(g1)
    requires IS_GLOBAL_ROOT(g0,x0)
    requires IS_GLOBAL_ROOT(g1,x1)
    ensures forall u:Node ::
      EXISTS_PATH(g0 union g1,x,u) <==>
        old( EXISTS_PATH(g0,x0,u) ) || old( EXISTS_PATH(g1,x1,u) )
    ensures forall u:Node, v:Node ::
      EDGE(g0 union g1,u,v) <==>
        old( EDGE(g0,u,v) ) || old( EDGE(g1,u,v) ) ||
        (u==last && v==x1)
    ensures ACYCLIC_LIST_SEGMENT(g0 union g1)
{
    last := x0
    while ( last.next != null )
        invariant CLOSED(g0)
        invariant ACYCLIC_LIST_SEGMENT(g0)
        invariant ACYCLIC_LIST_SEGMENT(g1)
        invariant IS_GLOBAL_ROOT(g0,x0)
        invariant IS_GLOBAL_ROOT(g1,x1)
        invariant last in g0
        invariant forall n:Node, m: Node ::
          {EDGE(g0, n, m)}{create_edge(n, m)}
          EDGE(g0, n, m) <==>
            old(EDGE(g0, n, m) &&
            create_edge(n, m) in EDGES(g0))
        invariant forall n:Node, m: Node ::
          {EDGE(g1, n, m)}{create_edge(n, m)}
          EDGE(g1, n, m) <==>
            old(EDGE(g1, n, m) &&
            create_edge(n, m) in EDGES(g1))
    {
        last := last.next
    }
    UPDATE(next, g0 union g1, last, x1)
    last.next := x1
    x := x0
}
```

Listing 7.2.: The `join` method for lists

```
forall n:Node, m: Node ::
  {EDGE(g0, n, m)}{create_edge(n, m)}
  EDGE(g0, n, m) <==>
    old(EDGE(g0, n, m) &&
    create_edge(n, m) in $$(g0))
```

Listing 7.3.: Invariant that the edges stay the same

`last`, such that `x` is the starting node of the list `g0 union g1`, and `last` is the node, which has been linked to `x1`.

The loop invariants should preserve that nothing has changed during the loop, as we do not update the graphs.

In the post state, the following properties should hold:

1. `g0` and `g1` remain acyclic.
2. `x` reaches a node `n`, if and only if `n` has been reachable by `x0` or `x1` in the prestate.
3. The edges of `g0 union g1` in the poststate should be nearly equivalent to the edges from `g0` and `g1` in the prestate, with the only difference that an edge between `last` and `x1` has been created.
4. `g0 union g1` is acyclic.

However, the frontend could not check that `g0 union g1` is acyclic. One essential loop invariant in the manual approach, which we did not use in the frontend, was that the edge sets of `g0` and `g1` remained the same, that means `$$(g0) == old($$(g0))` and `$$(g1) == old($$(g1))`. The reason why we did not use it, was because we wanted to avoid using the `$$` function. Adding these invariants suffices to prove acyclicity of `g0 union g1`. We tried to rewrite the invariants without using the `$$` function. We tried the following invariant:

```
forall n:Node, m: Node ::
          {EDGE(g0, n, m)}{create_edge(n, m)}
          EDGE(g0, n, m) <==>
            old(EDGE(g0, n, m)
```

But this was not sufficient. We ended up with the invariant for `g0` in Listing 7.3.

First of all, coming up with this invariant seems to be more difficult than coming up with `$$(g0) == old($$(g0))`. Secondly, here we not only use the `$$` function, but also `create_edge`, defined in section 2.1.3. Hence, it may be better to support reasoning about the edge set in the frontend.

Adding this loop invariant also sufficed to prove the third post condition.

```
ensures CLOSED(g0) && DISJOINT(g0, g1) ==> (
  forall u: Node, v: Node ::
    { exists_path($$(g0 union g1), u, v) }
    (u in g0) && (v in g1) ==>
    !exists_path($$(g0 union g1), u, v)
)
```

Listing 7.4.: One of the new postconditions to the framing method, for two graphs `g0` and `g1`. It specified that there is no path from `g0` to `g1`, if `g0` is closed and disjoint from `g1`. The other postcondition is analogous for `g1`.

Additionally, we were not able to fully prove the first property, namely that `g0` remained acyclic, while at the same time, we are now able to prove that `g0 union g1` is acyclic. Applying the framing wisdom in the end like in the previous example solved the problem.

The frontend could also not prove the second property, namely the fact that if there is a path from `x1` or `x0` to an arbitrary node `n` inside their corresponding graph in the prestate, then there must be a path from `x` to `n` in `g0 union g1` in the poststate. This again sounds like a framing problem. But applying the framing wisdom in the beginning and in the end of the method does not solve the problem.

On line 41 in the Listing 7.2, the frontend translates the `UPDATE` operation to a `UNLINK` and `LINK` operation. After updating the state of `g0 union g1` with `UNLINK`, the gained knowledge from applying the framing wisdom in the beginning gets lost. Hence, we need to apply framing after every state update. Here, we would have to additionally apply framing between unlinking and linking.

In the manual approach, the loop invariant `$$(g0) == old($$(g0))` was not sufficient for verifying that `g0 union g1` is acyclic. In addition to the invariant, the No Exit Wisdom defined in needed to be applied between unlinking and linking. In the frontend, we added two more postconditions to the `FRAMING` function, which work as a No Exit Axiom only for the two specified graphs. One of the postconditions is shown in Listing 7.4.

7.2.3. Other Test Cases

The other considered test cases did not need more manually added wisdoms.

The ring insert algorithm did not terminate. The reason was not clear. It was not able to show that the new graph in the post condition was also a list. Apparently, there are problems with the internal `create_node` function, and assigning the new ring to the `UNIVERSE`.

7.3. Conclusion

In this section, we will collect the gained experience by investigating in the previous test cases.

7.3.1. Automation

After having considered the test cases, it seems to be crucial to apply the framing wisdom, after every field update. More concretely, we should apply it after unlinking and linking. Additionally, it should be applied, whenever a fresh node has been generated, or a graph has been assigned.

The question is, on which graphs it should be applied. When a fresh node has been generated, we only need to apply the framing wisdom on the fresh node and on each existing graph.

For the other cases, we could apply it to every combination of two existing graphs, but this would not scale well, if we had multiple graphs, as the more framing wisdoms are applied, it takes more time to verify. In our test cases, we did not have big slowdowns because of framing axioms. But we assume that in most graph algorithms, the number of graphs is not very high.

That is why we will change the behaviour of our frontend, such that it can insert those wisdoms automatically, at every field update, fresh node generation, or graph assignment. However, it will only do it, if the user enables it via a magic field.

7.4. Comparative Analysis

We wanted to compare our tool with the one described in [2]. Unfortunately, it is not publicly available anymore.

8. Future Work

In this project, we implemented a frontend, which is able to verify several graph algorithms. Our goal was to design an extension that allows to verify graph algorithms without the lose of automation. While this goal was reached for small examples with few state updates, the extension reaches its limits in bigger graph algorithms, particularly in examples with several input graphs.

Hence, the frontend needs to be extended further. In the following sections, we will describe what we could do as next steps.

8.1. List Update Formulas

As seen in the example of splitting lists in section 7.2.1, the frontend was not able to prove that some `EXISTS_PATH` relations hold, when using the DAG update formulas, while it was possible in the manual approach with the List update formulas. Even though it might be a triggering problem, it could be beneficial to add Lists to our type system, because the list update formulas seem to be faster and easier to verify by the frontend.

8.2. Graph Destruction

In our frontend, we are able to create new graphs with method calls, by having graphs with access permissions. On the other hand, ever input graph of a method call, needs to return the access permissions in the post state of the method. We may want to add the possibility to pass graphs as input to methods, which take the access permissions in the prestate, but do not return them anymore.

8.3. Silicon

At the moment, we use Carbon as the backend to verify programs. We do not use Silicon, for several performance-related reasons. Mostly, the verification of the examples does not terminate at all. We should further investigate this problem and fix it.

8.4. More General Update Formulas

At the moment, the frontend does only support

8.5. Prioritized SMT Computations

In our frontend, we often inhale framing wisdoms. Generally, we could insert much more wisdoms. However, adding more wisdoms makes the code slower. In the verification tool used in [3], their heuristics only adds additional wisdoms if it is not possible to prove the postcondition.

In our heuristics, we add the framing axioms at points, where we assume that they are needed. It may make bigger graph algorithms faster, if we added them only, if we were not able to verify the program without them.

Appendices

A. Preamble

A.0.1. The Edge Domain

```
domain Edge {
    function edge_pred(e:Edge): Ref
    function edge_succ(e:Edge): Ref

    function create_edge(p:Ref, s:Ref): Edge
    function create_edge_(p:Ref, s:Ref): Edge

    axiom edge_injectivity {
        forall p:Ref, s:Ref ::
            { create_edge(p,s) }
                edge_pred( create_edge(p,s) ) == p &&
                edge_succ( create_edge(p,s) ) == s
    }
}
```

A.0.2. The TrClo Domain

```
domain TrClo {

    function exists_path(EG:Set[Edge], start:Ref, end:Ref): Bool
    function exists_path_(EG:Set[Edge], start:Ref, end:Ref): Bool
    function exists_spath(EG:Set[Edge], from:Set[Ref], to:Ref): Bool

    /// U is the universe;
    /// EG is the edge graph (specifies the edge relation)
    /// A is the color predicate
    /// M is the marker
    function apply_noExit(EG:Set[Edge], U:Set[Ref], M:Set[Ref]): Bool

    /** axiomatization of the set for instantiating color axioms with
    ↪  unary reachability */
    function inst_uReach(EG:Set[Edge], x:Ref): Set[Ref]
    function inst_uReach_rev(EG:Set[Edge], x:Ref): Set[Ref]
```

```

/** The properties of the edge relation. */
function acyclic_graph(EG:Set[Edge]): Bool
function unshared_graph(EG:Set[Edge]): Bool
function func_graph(EG:Set[Edge]): Bool


function edge(EG:Set[Edge], p:Ref, s:Ref): Bool
function edge_(EG:Set[Edge], p:Ref, s:Ref): Bool

function succs(EG:Set[Edge], pred:Ref): Set[Ref]

axiom ax_Succs {
    forall EG:Set[Edge], pred:Ref, succ:Ref ::
        { succ in succs(EG,pred) }
            succ in succs(EG,pred) <==> edge_(EG,pred,succ)
}

axiom ax_EdgeSynonim {
    forall EG:Set[Edge], p:Ref, s:Ref ::
        { edge(EG,p,s) }
            edge(EG,p,s) <==> edge_(EG,p,s)
}
axiom ax_Edge {
    forall EG:Set[Edge], p:Ref, s:Ref ::
        { create_edge(p,s) in EG }
        { edge(EG,p,s) }
            edge_(EG,p,s) <==> create_edge(p,s) in EG
}

// The first color axiom from the paper. (Via Refs)
axiom ax_NoExit {
    forall EG:Set[Edge], U:Set[Ref], M:Set[Ref] :: {
    ↪  apply_noExit(EG,U,M) }

    ( apply_noExit(EG,U,M) ==> (

        ( forall u:Ref, v:Ref :: { edge(EG,u,v) }   (u in M) && (v in
        ↪  U) && !(v in M) ==> !edge(EG,u,v) ) ==>
        ( forall u:Ref, v:Ref :: { exists_path(EG,u,v) }(u in M) &&
        ↪  (v in U) && !(v in M) ==> !exists_path(EG,u,v) )
    ) )
}

```

```
    axiom ax_instantiation_uReach {
        forall EG:Set[Edge], x:Ref, v: Ref ::
            { v in inst_uReach(EG,x) }
            { exists_path(EG,x,v) }
                v in inst_uReach(EG,x) <==> exists_path(EG,x,v)
    }

    axiom ax_instantiation_uReach_rev {
        forall EG:Set[Edge], u: Ref, y:Ref ::
            { u in inst_uReach_rev(EG,y) }
            { exists_path(EG,u,y) }
                u in inst_uReach_rev(EG,y) <==> exists_path(EG,u,y)
    }

    axiom ax_Alias {
        forall EG:Set[Edge], start:Ref, end:Ref ::
            { exists_path(EG,start,end) }
                exists_path(EG,start,end) <==> exists_path_(EG,start,end)
    }

    /** T1 -- Ref-oriented encoding */
    axiom ax_ExistsPath {
        forall EG:Set[Edge], start:Ref, end:Ref ::
            { exists_path(EG,start,end) }
            { edge(EG,start,end) }
                exists_path_(EG,start,end) <==>
                    start == end
                    || !(forall w:Ref ::
                        { edge(EG, start, w)}{ exists_path(EG, w, end) }
                            !(edge(EG,start,w) &&
                            ↪  exists_path_(EG,w,end)))
//                      || exists w:Ref :: edge(EG,start,w) &&
↪   exists_path_(EG,w,end)
    }

    /** Follows from T1 && IND */
    axiom ax_ExistsPathTrans {
        forall EG:Set[Edge], u:Ref, v:Ref, w:Ref ::
            { exists_path(EG,u,w), exists_path(EG,w,v) }
                exists_path_(EG,u,w) && exists_path_(EG,w,v) ==>
                ↪  exists_path_(EG,u,v)
    }

    axiom ax_AcyclicGraph {
        forall EG:Set[Edge] ::
            { acyclic_graph(EG) }
                acyclic_graph(EG) <==>
                    forall v1:Ref, v2:Ref ::
                        { edge(EG,v1,v2) }
                        { exists_path(EG,v2,v1) }
                            !edge(EG,v1,v2) || !exists_path(EG,v2,v1)
    }

    axiom ax_UnsharedGraph {
        forall EG:Set[Edge] ::
            { unshared_graph(EG) }
                unshared_graph(EG) <==>
                    forall v1:Ref, v2:Ref, v:Ref ::
                        { edge(EG,v1,v), edge(EG,v2,v) }
                            edge(EG,v1,v) && edge(EG,v2,v) ==> v1 == v2
    }

    axiom ax_FuncGraph {
        forall EG:Set[Edge] ::
            { func_graph(EG) }
                func_graph(EG) <==>
                    forall v1:Ref, v2:Ref, v:Ref ::
                        { edge(EG,v,v1), edge(EG,v,v2) }
                            edge(EG,v,v1) && edge(EG,v,v2) ==> v1 == v2
    }

    axiom ax_ExistsSetPath {
        forall EG:Set[Edge], from:Set[Ref], to:Ref ::
            { exists_spath(EG,from,to) }
                exists_spath(EG,from,to) <==>
                    !(forall f:Ref ::
                        {f in from}{exists_path(EG, f, to)}
                            !(f in from && exists_path(EG,f,to)))
        }
}
```

A.0.3. The ZOPG Domain

A.0.4. The apply_TCFraming Function

```
function apply_TCFraming(g0: Set[Ref], g1: Set[Ref]): Bool
  requires (forall n_0: Ref :: { n_0.next } (n_0 in g0) ==> acc(n_0.next,
  ↪  1 / 2))
  requires (forall n_0: Ref :: { n_0.next } (n_0 in g1) ==> acc(n_0.next,
  ↪  1 / 2))
  ensures g0 union g1 == g1 union g0
  ensures (forall u: Ref, v_0: Ref :: { exists_path($$(g0), u, v_0) } {
  ↪  exists_path($$(g0 union g1), u, v_0) } (u in g0) && ((v_0 in g0) &&
  ↪  exists_path($$(g0), u, v_0)) ==> exists_path($$(g0 union g1), u,
  ↪  v_0))
  ensures (forall u: Ref, v_0: Ref :: { exists_path($$(g1), u, v_0) } {
  ↪  exists_path($$(g1 union g0), u, v_0) } (u in g1) && ((v_0 in g1) &&
  ↪  exists_path($$(g1), u, v_0)) ==> exists_path($$(g1 union g0), u,
  ↪  v_0))
  ensures (forall u: Ref, v_0: Ref :: { exists_path($$(g0), u, v_0) } {
  ↪  exists_path($$(g0 union g1), u, v_0) } (u in g0) && ((v_0 in g0) &&
  ↪  !exists_path($$(g0 union g1), u, v_0)) ==> !exists_path($$(g0), u,
  ↪  v_0))
  ensures (forall u: Ref, v_0: Ref :: { exists_path($$(g1), u, v_0) } {
  ↪  exists_path($$(g1 union g0), u, v_0) } (u in g1) && ((v_0 in g1) &&
  ↪  !exists_path($$(g1 union g0), u, v_0)) ==> !exists_path($$(g1), u,
  ↪  v_0))
  ensures (forall u: Ref, v_0: Ref :: { exists_path($$(g0), u, v_0) } {
  ↪  exists_path($$(g0 union g1), u, v_0) } (u in g0) && ((v_0 in g0) &&
  ↪  (!exists_path($$(g0), u, v_0) && ((CLOSED(g0) || CLOSED(g1)) &&
  ↪  DISJOINT(g0, g1)))) ==> !exists_path($$(g0 union g1), u, v_0))
  ensures (forall u: Ref, v_0: Ref :: { exists_path($$(g1), u, v_0) } {
  ↪  exists_path($$(g0 union g1), u, v_0) } (u in g1) && ((v_0 in g1) &&
  ↪  (!exists_path($$(g1), u, v_0) && ((CLOSED(g0) || CLOSED(g1)) &&
  ↪  DISJOINT(g0, g1)))) ==> !exists_path($$(g1 union g0), u, v_0))
  ensures CLOSED(g0) && DISJOINT(g0, g1) ==> (forall u: Ref, v_0: Ref ::
  ↪  { exists_path($$(g0 union g1), u, v_0) } (u in g0) && (v_0 in g1)
  ↪  ==> !exists_path($$(g0 union g1), u, v_0))
  ensures CLOSED(g1) && DISJOINT(g0, g1) ==> (forall u: Ref, v_0: Ref ::
  ↪  { exists_path($$(g0 union g1), v_0, u) } (u in g0) && (v_0 in g1)
  ↪  ==> !exists_path($$(g0 union g1), v_0, u))
  ensures IS_ZOPG(g0 union g1) ==> IS_ZOPG(g0)
```

A.0.5. The DAG Update Formulas

```
method unlink_DAG_$field$(g: Set[Ref], x:Ref)
    requires x in g
    requires PROTECTED_GRAPH(g,x,$field$)
    ensures PROTECTED_GRAPH(g,x,$field$)
    ensures x.$field$ == null
    ensures old(x.$field$) == null ==> $$(g) == old($$(g))
    ensures old(x.$field$) != null ==> forall v1:Ref, v2:Ref ::
        { edge($$(g),v1,v2) }
            edge($$(g),v1,v2) <==> edge(old($$(g)),v1,v2) && !(v1==x &&
            ↪  v2==old(x.$field$))
    ensures old(x.$field$) != null ==> forall v1: Ref, v2: Ref ::
        {exists_path($$(g), v1, v2)}
            !exists_path(old($$(g)), v1, v2)  ==> !exists_path($$(g), v1,
            ↪  v2)
    ensures old(x.$field$) != null ==> forall v1:Ref, v2:Ref ::
        { exists_path($$(g),v1,v2) }
             exists_path($$(g),v1,v2) <==> (v1==v2) ||

                ( exists n:Ref :: exists_path(old($$(g)),v1,n) &&
                ↪  exists_path(old($$(g)),n,v2) &&

                    ( !exists_path(old($$(g)),n,x) &&
                    ↪  !exists_path(old($$(g)),x,n) ||

                      !exists_path(old($$(g)),n,old(x.$field$)) &&
                      ↪  !exists_path(old($$(g)),old(x.$field$),n) )
                )
                ||
                ( exists n:Ref :: n != x && n != old(x.$field$) &&
                ↪  exists_path(old($$(g)),x,n) &&
                ↪  exists_path(old($$(g)),n,old(x.$field$)) )
                ||

                ( exists u:Ref, v:Ref :: (u != x || v != old(x.$field$))
                ↪  && edge(old($$(g)),u,v) &&
                ↪  exists_path(old($$(g)),v1,u) &&

                                                                                    ↪  exists_pat
                                                                                    ↪  &&
                                                                                    ↪  exists_pat
                                                                                    ↪  &&
                                                                                          ↪  exists_pat
                                                                                          ↪  )

function update_DAG_invariant(g:Set[Ref], x:Ref, y:Ref): Bool
    requires PROTECTED_GRAPH(g)
{
    !(exists_path($$(g), y, x))
}


method link_DAG_$field$(g:Set[Ref], x:Ref, y:Ref)
    requires x in g
    requires y != null ==> y in g && y != x
    requires PROTECTED_GRAPH(g,x,$field$)
    ensures PROTECTED_GRAPH(g,x,$field$)
    ensures x.$field$ == y
    ensures y == null ==> $$(g) == old($$(g))
    ensures y != null ==> forall v1:Ref, v2:Ref ::
        { edge(old($$(g)),v1,v2) }
            edge($$(g),v1,v2) <==> edge(old($$(g)),v1,v2) || (v1==x &&
            ↪  v2==y)
    ensures y != null ==> forall v1:Ref, v2:Ref ::
        { exists_path($$(g),v1,v2) }
            exists_path($$(g),v1,v2) <==> exists_path(old($$(g)),v1,v2)
            ↪  || (exists_path(old($$(g)),v1,x) &&
            ↪  exists_path(old($$(g)),y,v2))
    ensures acyclic_graph($$(g))
```

A.0.6. The ZOPG Update Formulas

B. Declaration of originality

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Deductive Verification of Imperative Graph Algorithms

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Sivanrupan	Gishor

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Siebnen, 15.09.2018	S. Gishor

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

Bibliography

[1] G. Dong and J. Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Information and Computation*, 120(1):101 – 106, 1995.

[2] S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, pages 756–772, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[3] T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In R. Nieuwenhuis, editor, *Automated Deduction – CADE-20*, pages 99–115, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[4] A. Ter-Gabrielyan. Graph reachability examples in viper. Accessed: 2018-09-15.