Shape invariants : formalization of transfer functions

Internship at IMDEA
Supervisor : Laurent Mauborgne

Florence Clerc

From April to August 2011

Abstract

Checking if some properties are respected throughout the execution of a program is one of the key question within abstract interpretation is the question of. We have focused on static analysis and we propose a new efficient way of finding shape invariants. Our approach is based on shape graphs on which a few operations are performed. It should be less ressource-consuming and a lot more intuitive than TVLA although there is still some tests to be done.

This internship has been about formalizing ideas and proving the validity of our methods. We have the algorithms for most of the transfer functions but there is still some work to be done on implementing them, detailing the other transfer functions and digging further in the theory.

I would like to thank M.Mauborgne for accepting me as an intern, for answering all my questions and for letting me dig in (sometimes) weird and unexpected directions.

I would also like to thank Mme Delais, M.Fouques and M.Cousot for this year at the École Normale Supérieure of Paris.

1 Introduction

1.1 Context

Software is present in our everyday life and our safety depends increasingly on its reliability. For that reason, proving that a piece of code has the expected properties has become critical. Whereas people used to run critical code on all possible scenarios, verifying it beforehand has proven very efficient.

Many approaches relying on abstract interpretation [7] [6] have been developped such as model-checking where the system that we want to check is represented by a Kripke structure [1], but we focused on static analysis and the search of shape invariants. We are looking for a structure that would give us information on how a piece of code is executed and what happens in the memory of the computer executing it. There already exists a method : the Three-Valued Logic Analysis [14]. TVLA represents the execution of a program by a graph where the nodes represent sets of memory locations and the edges possible transitions. The main idea of TVLA is to evaluate predicates on each nodes of this graph. Out of these predicates, there is one which represents whether a node has been obtained by summing up other ones. Both this approach and ours need at one point to "focus" on a given path so that we arrive on a node representing a single memory location. But in order to do this, we may split the graph into many others so that the path leads deterministically to a single location for each graph. This operation may be very costly as it may create an enormous number of graphs : one only has to visualize the case where the graph is complete. Moreover, in the TVLA approach, all properties are evaluated on all nodes, which may increase the complexity.

On the contrary, our approach is very intuitive and visual and it should be a lot more efficient than TVLA as we have developped a *union*function which is a possible answer to the problem of the number of graphs that may be created. This function should also be able to reduce the size of a graph by noticing nodes that are looking very similar, which would also help reduce the complexity.

1.2 Concrete and Abstract Semantics

1.2.1 Concrete Semantics: Heaps and Procedures

Heaps The concrete semantics has two objects : a stack containing values of program variables and a heap which can be seen as a set of disjoint memory locations. Accesses outside of those locations are abstracted by a total loss of information (including error state).

In that setting, the value of a variable can be either a numerical value, a valid location, or the special location **null** which does not correspond to any valid location. We define *Loc* a set of memory locations,

$Fields$ a set of names, $Vars$ a set of variables and Val a set of basic values such as integers plus floats plus booleans...

A *memory state* is a partial function that maps each couple of memory location and field name to either a memory location, a basic value, or the special value **null** . **null** corresponds to a fictive memory location that cannot be in the domain of a memory state and stands for an unallocated memory location. We write $MS = Loc \times Fields \to Loc \cup Val \cup \{\textbf{null}\}$ for the set of all possible memory states.

We suppose that Loc contains at least a special field $Stack$ containing the location of the top of the call stack. A *path* will be a possibly empty sequence of names.

One may refer to [10] for the mathematical definition or to section 3.1.

Program Executions A program is a list of procedures. Each procedure is given by a name, a set of arguments, local variables and a sequence of instructions. Each instruction is given a unique label.

A run starting in a given memory state M for a procedure consists in the sequence of memory states starting with M and following the flow of instructions of that procedure.

The concrete semantics of a program with a given starting procedure is the set of all runs from any valid memory state for that procedure. The notion of "valid" memory state might depend on a type attached to each memory location.

1.2.2 Shape Graphs

In order to abstract the concrete heap, we are using shape graphs. We first introduce the sets $Node$ and $Edge$ which are the sets of all nodes and the set of all edges. A *Shape Graph* G is a tuple (G_N, G_E, G_π, c_G) where

- $G_N \subset Node$ is the set of nodes of G. A node represents one or many locations.
- $G_E \subset Edge$ is the set of edges of G. An edge is a triple (n_0, n_1, f) with $n_0, n_1 \in G_N$ and f is a label in $Fields \cup Vars$. It represents the possible transitions from one location represented by n_0 to another one represented by n_1.
- G_π is a set of properties : $\mathcal{P}(G_N) \to \mathcal{P}(MS)$.
- $c_G : G_N \cup G_E \to \mathcal{P}(\mathbb{N})$ is the cardinality of a node or of an edge. It is defined on all nodes and all edges of G. For a node n, $c_G(n)$ represents the possible number of locations corresponding to n. For an edge e, $c_G(e)$ represents the possible number of transitions that exist with a given label from a location represented by a given node to another one. We will not show that this property is verified throughout the construction of the invariant but we will let the reader convince himself that it is.

Let us now define the notion of embedding :

Definition 1. *A memory state m is embedded in a shape graph G (and we write $m \tilde{\in} G$ or $m \tilde{\in}_\phi G$) if there is a mapping ϕ from Loc to nodes of G such that :*

- *for all couple (l, f) in the domain of m, if $m(l, f)$ is a location, then $(\phi(l), \phi(m(l, f)), f)$ is an edge of the abstract graph,*
- *and for all $N \subset G_N$ in the domain of G_π, m restricted to $\phi^{-1}(N)$ is in $G_\pi(N)$.*

In order to have an efficient algorithm, it is important to keep things local as much as possible. So, we will nearly always attach properties to single nodes but as little as possible to sets of nodes.

1.3 Our approach

This is only a very brief presentation of the ideas and the use we will make of each function so that the reader may better understand the developped algorithms. The key behind this idea is the use of shape graphs. We are designing instruction by instruction the structure we get after executing the piece of code (program or procedure). To that purpose, we define some transfer functions that will allow us to manipulate graphs :

- $focus$allows us to select the location where we "arrive" by "following a path" by selecting the corresponding node. The $focus$operation expands graphs and may create a large amount of graphs. This function also exists in TVLA and is responsible for its huge cost.
- *assign*allows us to modify or add edges to specific locations which corresponds to an assignment in the piece of code.

- *union*allows us to union two graphs with a chosen degree of precision. The *union*operation allows us to group similar graphs, which may let us have better performances than the TVLA approach. Moreover, our approach is very visual and therefore very easy to work with. It has been designed so that it may also be used to compare a graph with himself and therefore reduce his size. Moreover, the result of this function depends on a precision parameter given by the person testing the code.

One transfer function has not been yet formalized : we still are not able to check if a graph is a subgraph from another one. We will need this function in order to stop the iterations when checking loops.

2 Formalization of transfer functions

2.1 Preliminary functions and definitions

When a variable x is assigned to a node n, there exists an edge $(Stack, n, x)$ where $Stack$ is a node with no incoming edges and where all outcoming edges have a label in the variables set $Vars$.

$\mathcal{C}$ is the set of all allowed cardinalities, containing at least $\{1\}$. It is partially ordered, with the order being the inclusion ordering and has a top element $\mathbb{N}$. Moreover,

$$\forall G \in Graph\ \forall n \in G_N\ \forall e \in G_E\ c_G(n) \in \mathcal{C}\ \wedge\ c_G(e) \in \mathcal{C}$$

Let us also clarify $A \bigoplus B$ where $A, B \subset \mathbb{N}$: $A \bigoplus B = \{a+b | a \in A\ \wedge b \in B\}$, and $A \bigotimes B$ where $A, B \subset \mathbb{N}$: $A \bigotimes B = \{a \times b | a \in A\ \wedge b \in B\}$.

We will note $c_G(n)$ the cardinality of the n node in G and $c_G(e)$ the cardinality of the e edge in G.

We suppose the following functions exist :

- $loop : Edge \rightarrow bool$ which expresses whether an edge is a self-loop
- $duplicate : Graph \rightarrow Graph$ which takes a shape graph and copies it
- $intensive : Property \rightarrow bool$ which expresses whether having a property respected by some nodes individually implies having the property respected by the set of these nodes

We suppose that a few methods on graphs exist :

- $deleteNode : Node \rightarrow ()$ which suppresses a node from a graph, removes all the edges that where linked to it and removes it from the properties it was associated to
- $deleteNodes : \mathcal{P}(Node) \rightarrow ()$ which suppresses a set of nodes from a graph using the $deleteNode$ method
- $newNode : () \rightarrow Node$ which creates a new node
- $addNode : Node \times cardinality \rightarrow ()$ which adds the node in $\mathcal{G}_N$, with the given cardinality
- $addEdge : Edge \times \mathcal{C} \rightarrow ()$ which adds a new edge in $\mathcal{G}_E$ with the given cardinality
- $addEdges : \mathcal{P}(Edge \times \mathcal{C}) \rightarrow ()$ which adds all the new edges in $\mathcal{G}_E$ with the given cardinality
- $removeEdges : \mathcal{P}(Edge) \rightarrow ()$ which deletes all the corresponding edges in the graph
- $properties : Node \rightarrow \mathcal{P}(Property)$ which gives all the properties where a given node appears
- $edges : Node \rightarrow \mathcal{P}(Edge \times cardinality)$ which gives all the edges going to or from a given node with their cardinality

2.2 Defining focus

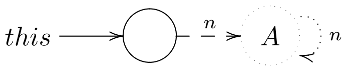
2.2.1 Explaining the idea

In a word, the aim of $focus$is to obtain a graph such that whatever operation we compute, only one location is taken into account rather then a set of locations. Let us now exhibit an example where the focus operation allows to keep some information. We consider $\mathcal{C} = \{\{1\}, \{0, 1\}, \mathbb{N}\}$.

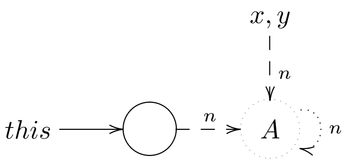
```
class List {
   List n;

   bool next(){
      x = this.n ;
      y = x.n ;
      return (x == y) ;
   }
}
```

We want to have an idea of when `next()` may return `true`. We assume that lists are acyclic. We will draw shape graphs with the convention that dotted nodes and edges are associated with no cardinality property (which is the same as $\mathbb{N}$), plain ones with exactly one cardinality property, and dashed ones with cardinality zero or one. The acyclic property will be represented by writing A on the corresponding nodes.

Before defining `x` and `y`, we have the following :

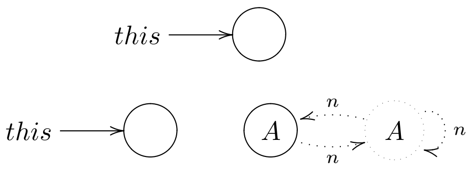


If we would not perform the focus operation, we would obtain :

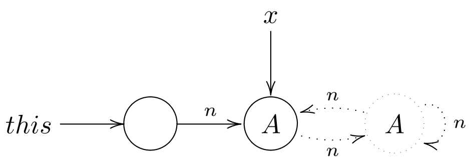


This does not give us the information we are looking for. The focus operation allows us to expand the node as we need to, and we would obtain the three following graphs when focusing on *this.n* and assigning x:

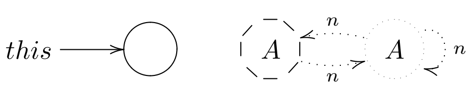
The two following graphs will assign to x :



The following graph allows us to assign x to a single node representing a single location :

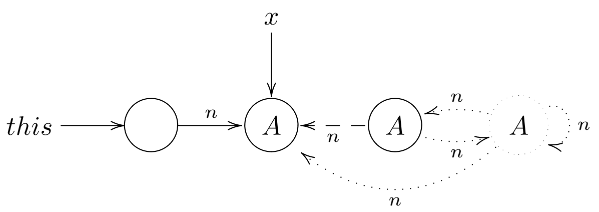


Please note that the self-loops on the nodes created of cardinality 1 have been erased as they would not match the property of acyclicity. You may also note that the two first graphs may be unioned in a single graph.

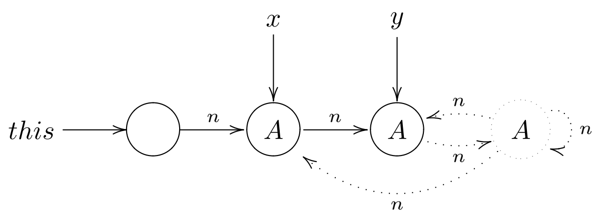


Once this is obtained, we may focus on $x.n$ in order to assign y. We only do this on the graph that has not returned an error, it would give two graphs :

This one would assign null to y :



This one returns no error :



In both graphs we have erased the edges that did not respect the acyclicity property. In the second graph we have erased the edge of cardinality $\{0,1\}$ from the y-assigned node to the x-assigned one as there was one edge of cardinality 1 in the other way and it would have created a loop otherwise.

This allows us to conclude that as long as we have an acyclic list of length at least three, the list does not loop between the second and the third element.

2.3 Formalizing

Let us now define formally our algorithm. We first define a distance on $\mathcal{P}(\mathbb{N})$:

$$d(u,v) = |\{x \in u \setminus v\} \cup \{y \in v \setminus u\}| = |(u \cup v) \setminus (u \cap v)|$$

This allows us to define :
$d_{min}(card) = min\{d(card, \bigoplus_{j=0}^{k} c_j) | \forall j \ c_j \in \mathcal{C} \ \wedge \ c_0 = \{1\} \ \wedge card \subset \bigoplus_{j=0}^{k} c_j\}$.
$S(card) = \Big\{\{c_0, ..., c_k\} | c_0, ..., c_k \in \mathcal{C} \ \wedge c_0 = \{1\} \ \wedge card \subset \bigoplus_{j=0}^{k} c_j \ \wedge d(card, \bigoplus_{j=0}^{k} c_j) = d_{min}(card)\Big\}$
$k_{min}(card) = min \{k | c_0, ..., c_k \in S(card)\}$

We may now define the function $cardinalityNewNode : \mathcal{C} \rightarrow int \times \mathcal{P}(\mathcal{C})$:

```
let $cardinalityNewNode(card)$ =
```

$$\begin{cases} (0, \{\{1\}\}) & \text{if } card = \{1\} \\ (k, \{c_0, ..., c_k\}) & \{c_0, ..., c_k\} \in S(card) \\ & \text{ where } k = k_{min}(card) \text{ and where } c_k\text{'s upper bound is the highest of all } c_j \\ & \text{ and (if necessary) where } c_k \text{ bears the worst imprecision of all } c_j \end{cases}$$

We also define $propUpdate : \mathcal{P}(Property) \times \mathcal{P}(Node \times \mathcal{P}(Node)) \times Graph \rightarrow Graph$:

```
let $propUpdate(prevProp, nodesAdded, G)$ =
  let $newProperties$ =
```

$$\begin{cases} \mathcal{P}(G_N) \rightarrow & \mathcal{P}(MS) \\ N' \mapsto & prevProp(N') \text{ if } \forall n \in N' \ \neg(\exists A \ \exists m \ n \in A \ \wedge (m, A) \in nodesAdded) \\ N' \mapsto & prevProp(N) \text{ where } N = N' \cup \{n | \exists A \ (n, A) \in nodesAdded\} \\ & \setminus \{m | \exists n \ \exists A \ (n, A) \in nodesAdded \ \wedge m \in A\} \text{ and if } \neg intensive(prevProp|_N) \\ \{m\} \mapsto & prevProp(n) \text{ if } \exists A \ (n, A) \in nodesAdded \ \wedge m \in A \ \wedge intensive(prevProp|_{\{n\}}) \end{cases}$$

```
  let $G_\pi$ = $newProperties$
  return $G$
```

And we are finally defining the method on a shape graph G : $cardUpdate : Node \times Node \times \mathcal{C} \rightarrow \mathcal{C}$

```
let $cardUpdate(a, b, card)$ =
  let $tempCard$ = $card \cap (c_G(a) \bigotimes c_G(b))$
  let $returnCard$ = $c \in \mathcal{C}$ such that $tempcard \subset c$ and $\forall p \in \mathcal{C} \ (d(tempCard, c) < d(tempCard, p))$
$\vee \big((d(tempCard, c) = d(tempCard, p)) \wedge (minc < minp)\big)$
  return $returnCard$
```

We are now going to define $focus : Graph \times Fields^* \rightarrow \mathcal{P}(Graph)$

```
let rec $focus(G, c)$ =
  if $c = \epsilon$ then return ($\{G\}$)
  else
```

```
        we may write $c$ as $p.f$.
        let $ensGraph = focus(G,p)$
        let $graphToReturn = \emptyset$
        for $G'$ in $focus(G,p)$ :
            let $(graphTemp, stack_p, stack_{p.f}, prevProp, edgesToTreat = focusNodes(G',p,f)$
            for $(G'', nodesAdded)$ in $graphTemps$
                let $stackTemp_p = stack_p \cap G''_N$
                let $stackTemp_{p.f} = stack_{p.f} \cap G''_N$
                let $edgesTemp = edgesToTreat \cap (G''_N \times G''_N \times Fields)$
                $graphToReturn$= $graphToReturn \cup$
$focusEdges(G'', p.f, stackTemp_p, stackTemp_{p.f}, edgesTemp, nodesAdded)$
            end for
        end for
    end if
    return $graphToReturn$
```

The $focusNodes : Graph \times Fields^* \times Fields \rightarrow \mathcal{P}(Graph) \times \mathcal{P}(Node) \times \mathcal{P}(Node) \times \mathcal{P}(Property) \times \mathcal{P}(Edges)$ procedure treats the node for a step of recurence of $focus$, it first separates the cases where a node can be of nul cardinality, then expands the nodes with the procedure $cardinalityNewNode$:

```
let $focusNodes(G,p,f)$ =
    let $stack_p = \{n | n$ is reachable through $p$ in $G\}$
    let $stack_{p.f} = \{n | \exists m \in stack_p$ such that $n \in m \star f\}$
    let $nodesZeroCard = \{n \in stack_{p.f} | 0 \in c_G(n)\}$
    let $nodesAdded = \emptyset$
    let $nodesToAdd = \{(n,k,\{c_0,...,c_k\}) | n \in stack_{p.f} \wedge (k,\{c_0,...,c_k\}) = cardinalityNewNode(c_G(n) \setminus \{0\})\}$
    let $edgesToTreat = \{(e, card) | e \in G.edges(n) \ \wedge \ n \in stack_{p.f} \ \wedge \ card = c_G(e)\}$
    let $graphToReturn = \emptyset$
    let $prevProp = G_\pi$
    let $ensGraph = \{G.deleteNodes(M) | M \subset nodesZeroCard\}$

    for $G'$ in $ensGraph$:
        for $(n,k,\{c_0...c_k\})$ in $nodesToAdd$ with $n \in G'_N$:
            for $i$ from $0$ to $k$ :
                let $node_{n,i} = G$.newNode()
                $G$.addNode($node_{n,i}, c_i$)
            end for
            $nodesAdded$.append(($n, \{node_{n,i} | i \in [\![0;k]\!]\}$))
            $G$.deleteNode($n$)
        end for
        $graphToReturn$= $graphToReturn \cap \{(G', \{(n,k,\{c_0...c_k\}) \in nodesToAdd | n \in G'_N\}\}$

        $G' = propUpdate(prevProp, nodesAdded, G')$
    end for
    return ($graphToReturn$, $stack_p$, $stack_{p.f}$, $prevProp$, $edgesToTreat$)
```

The $focusEdges : Graph \times Fields^* \times \mathcal{P}(Node) \times \mathcal{P}(Node) \times \mathcal{P}(Edges) \times \mathcal{P}(Node) \rightarrow \mathcal{P}(Graph)$ procedure corresponds to the focus procedure if we did not have to take into account the 0-cardinality cases.
Let us then explain the idea behind this procedure :

- we are first going to make a basis for all graphs that will be created. Therefore we first split the nodes that should be, then we add the edges that are common to all the graphs in $focus_G(c)$. While doing this, we are going to store the other edges in $differenciatingEdges$.
- we then create new graphs based on this common base to which we add one single edge or none from $differenciatingEdges$.

```
 let $focusEdges(G, p.f, stack_p, stack_{p.f}, edgesToTreat, nodesAdded)$ =
    let $nodesAdded = \emptyset$
    let $differenciatingEdges = \emptyset$
```

```
  let $nulCard = True$
  for $(e, card)$ in $edgesToTreat$ :
    we may write $e = (a, b, l)$
    let $A$ such that $(a, A) \in nodesAdded$ if such $A$ exists and $A = \{a\}$ otherwise
    let $B$ such that $(b, B) \in nodesAdded$ if such $B$ exists and $B = \{b\}$ otherwise
    if $loop(e)$ then
      if $l = f \wedge a \in stack_p$ then
        $G$.addEdges($\{((\alpha, \gamma, f), cardUpdate(\alpha, \gamma, card)) | \alpha \in A \setminus \{node_{a,0}\} \wedge \gamma \in A\}$)
        $differenciatingEdges$.append($(node_{a,0}, node_{a,0}, f)$)
        $differenciatingEdges$.append($(node_{a,0}, node_{a,1}, f)$)
        $nulCard = nulCard \wedge (0 \in card)$
      else
        $G$.addEdges($\{((\alpha, \gamma, l), cardUpdate(\alpha, \gamma, card)) | \alpha, \gamma \in A\}$)
      end if
    else
      if $a \in stack_p \wedge b \in stack_{p.f} \wedge l = f$ then
        $differenciatingEdges$.append( $(a, node_{b,0}, f)$ )
        $nulCard = nulCard \wedge (0 \in card)$
      else
        $G$.addEdges($\{((\alpha, \beta, f), cardUpdate(\alpha, \beta, card)) | \alpha \in A \wedge \beta \in B,)$
      end if
    end if
  end for
  for $e$ in $differenciatingEdges$ :
    $graphToReturn$.append(duplicate($G$).addEdge($e, 1$))
  end for
  if $nulCard$ then
    $graphToReturn$.append($G$)
  end if
  for $G'$ in $graphToReturn$:
    We erase the edges which do not correspond to the property associated to the set of nodes
it was previously on
  end for
  return $graphToReturn$
```

2.4 Defining Assign

We define $Label = Vars \cup Fields$. We will here assume that given an expression $p.f = r$ we have $p \in Label^*$, $f \in Label$ and $r \in Label^+ \cup Val$.

We define $assign : Graph \times expression \to \mathcal{P}(Graph)$

```
let $assign(G, p.f = r)$ =
  if $r \in Val$ then
    let $graphToReturn = assignVal(G, p.f = r)$
  else
    let $graphToReturn = assignLoc(G, p.f = r)$
  end if
  return $graphToReturn$
```

Where we define $assignLoc : Graph \times expression \to \mathcal{P}(Graph)$

```
let $assignLoc(G, p.f = r)$ =
  let $ensGraph = focus((focus(G, p)), r)$
  for $G'$ in $ensGraph$ :
    if there is a node reachable through $p$ then
      let $n_p$ be that node
      let $stack_r = \{n | n$ is reachable through $r$ in $G'\}$ /* of cardinality 1 or 0 */
      $G'$.removeEdges($\{(n_p, b, f) | b \in G'_N\}$)
      if $stack_r = \{n_r\}$ then
        $G'$.addEdge($(n_p, n_r, f), 1$)
      end if
    end if
    end for
return $ensGraph$
```

Where we define $assignVal : Graph \times expression \to \mathcal{P}(Graph)$

```
let $assignVal(G, p.f = r)$ =
    let $ensGraph = focus(G, p)$
    for $G'$ in $ensGraph$ :
        if there is a node reachable through $p$ then
            let $n_p$ be that node
            if there exists no node $n_r$ such that $G'.values(n_r) = \{r\}$ then
                $n_r = G'$.newNode()
                $G'$.addNode($n_r, 1$)
                $G'$.$values$($n_r, \{r\}$)
            end if
            let $n_r$ the node of $G'$ such that $G'.values(n_r) = \{r\}$
            $G'_E$.removeEdges($\{(n_p, n, f) | n \in G'_N\}$)
            $G'_E$.addEdge($(n_p, n_r, f), 1$)
        end if
    end for
return $ensGraph$
```

2.5 Defining union

2.5.1 Analyzing the similarity between two nodes of a graph

We intend to compare two graphs G^1 and G^2. For this, we are comparing each pair of nodes (N, M) where $N \in G^1_N$ and $M \in G^2_N$. Let us note $S(N, M)$ the function of comparison :

$$S(N,M) = \frac{1}{|\,Vars\,| + 2|\,Fields\,|} \times \left(\sum_{x \in Vars} \delta_{Vars}(N,M,x) + \sum_{f \in Fields} \delta_{incoming}(N,M,f) + \sum_{f \in Fields} \delta_{outcoming}(N,M,f) \right)$$

where :

$$\begin{aligned} \delta_{Vars} :& G^1_N \times G^2_N \times Vars && \to [0;1] \\ & (N,M,x) && \mapsto \neg \left([(S,N,x) \in G^1_E] \; xor \; [(S,M,x) \in G^2_E] \right) \end{aligned}$$

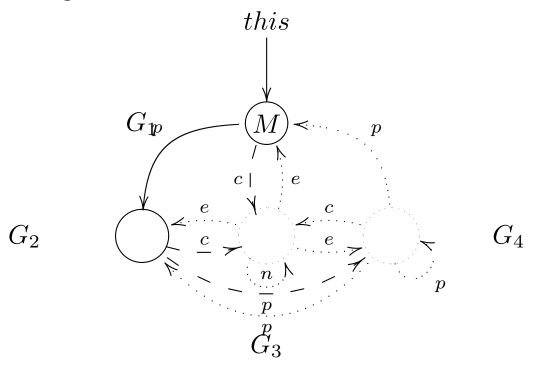
where :

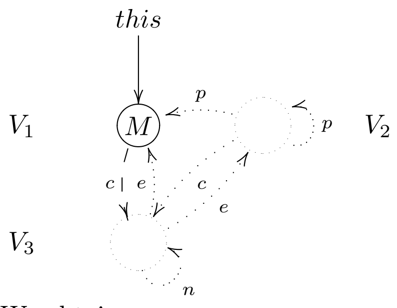
$$\begin{aligned} \delta_{incoming} :& G^1_N \times G^2_N \times Fields && \to [0;1] \\ & (N,M,f) && \mapsto \frac{1}{|S_N| \times |S_M|} \sum_{N_0 \in S_N, M_0 \in S_M} S(N_0, M_0) \\ & && \text{where } S_N = \{N_0 | (N_0, N, f) \in G^1_E\} \text{ and } S_N = \{M_0 | (M_0, M, f) \in G^1_E\} \\ & 1 && \text{if } S_N = \emptyset \wedge S_M = \emptyset \\ & 0 && \text{if } S_N = \emptyset \; xor \; S_M = \emptyset \end{aligned}$$

and where :

$$\begin{aligned} \delta_{outcoming} :& G^1_N \times G^2_N \times Fields && \to [0;1] \\ & (N,M,f) && \mapsto \frac{1}{|S_N| \times |S_M|} \sum_{N_0 \in S_N, M_0 \in S_M} S(N_0, M_0) \\ & && \text{where } S_N = \{N_0 | (N, N_0, f) \in G^1_E\} \text{ and } S_N = \{M_0 | (M, M_0, f) \in G^1_E\} \\ & 1 && \text{if } S_N = \emptyset \wedge S_M = \emptyset \\ & 0 && \text{if } S_N = \emptyset \; xor \; S_M = \emptyset \end{aligned}$$

This gives us a "matrix". Let us consider the following example :





We obtain :

$$\begin{array}{c} \begin{array}{cc} & \begin{array}{ccc} V_1 & V_2 & V_3 \end{array} \\ \begin{array}{c} G_1 \\ G_2 \\ G_3 \\ G_4 \end{array} & \begin{pmatrix} S(V_1,G_1) & S(V_2,G_1) & S(V_3,G_1) \\ S(V_1,G_2) & S(V_2,G_2) & S(V_3,G_2) \\ S(V_1,G_3) & S(V_2,G_3) & S(V_3,G_3) \\ S(V_1,G_4) & S(V_2,G_4) & S(V_3,G_4) \end{pmatrix} \end{array} \\ = \end{array}$$

$$\begin{array}{c|ccc} & V_1 & V_2 & V_3 \\ \hline G_1 & \frac{1}{9}(5+S(V_2,G_4)+2S(V_3,G_3)) & \begin{array}{c} \frac{1}{9}\Big(4+2S(V_3,G_3)+ \\ \frac{1}{2}(S(V_2,G_2)+S(V_1,G_2))\Big) \end{array} & 0 \\ G_2 & \begin{array}{c} \frac{1}{9}\Big(4+2S(V_3,G_3)+ \\ +\frac{1}{2}(S(V_2,G_1)+S(V_2,G_4))\Big) \end{array} & \begin{array}{c} \frac{1}{9}\Big(5+2S(V_3,G_3)+\frac{1}{2}S(V_2,G_1) \\ +\frac{3}{2}S(V_2,G_4)\Big) \end{array} & \frac{1}{9} \\ G_3 & \frac{1}{9} & \frac{1}{9} & \begin{array}{c} \frac{1}{9}\Big(5+2S(V_3,G_3)+ \\ \frac{2}{6}\Big(S(V_1,G_1)+S(V_1,G_2)+S(V_1,G_4) \\ +S(V_2,G_1)+S(V_2,G_2)+S(V_2,G_4)\Big)\Big) \end{array} \\ G_4 & \begin{array}{c} \frac{1}{9}\Big(4+2S(V_3,G_3)+ \\ \frac{1}{2}(S(V_2,G_2)+S(V_2,G_4))\Big) \end{array} & \begin{array}{c} \frac{1}{9}\Big(5+2S(V_3,G_3)+\frac{2}{3}S(V_2,G_4)+\frac{2}{3}S(V_2,G_2) \\ \frac{1}{6}\Big(S(V_1,G_1)+S(V_1,G_2)+S(V_1,G_4) \\ S(V_2,G_1)+S(V_2,G_2)+S(V_2,G_4)\Big)\Big) \end{array} & \frac{1}{9} \end{array}$$

We are not going to solve the previous system of equations. But knowing that $S(N,M) \in [0;1]$, we have the following intervals :

$$\begin{array}{cc} & \begin{array}{ccc} V_1 & V_2 & V_3 \end{array} \\ \begin{array}{c} G_1 \\ G_2 \\ G_3 \\ G_4 \end{array} & \begin{pmatrix} \left[\frac{5}{9},\frac{8}{9}\right] & \left[\frac{4}{9},\frac{7}{9}\right] & \{0\} \\ \left[\frac{4}{9},\frac{7}{9}\right] & \left[\frac{5}{9},1\right] & \left\{\frac{1}{9}\right\} \\ \left\{\frac{1}{9}\right\} & \left\{\frac{1}{9}\right\} & \left[\frac{5}{9},1\right] \\ \left[\frac{4}{9},\frac{7}{9}\right] & \left[\frac{5}{9},1\right] & \left\{\frac{1}{9}\right\} \end{pmatrix} \end{array}$$

We may iterate by replacing $S(N,M)$ by the intervals we have just computed, this allows us to get intervals that become very narrow without solving the system of equations we got. This is the idea that is developped in the following algorithms.

Given that, we obtain that we should choose $V_1 \equiv G_1$, $V_2 \equiv G_2 \equiv G_4$ and $V_3 \equiv G_3$ which gives back 2_1

2.5.2 Using this

We are going to compare two graphs G^1 and G^2. We develop two ways of doing it, one is the most accurate but is in space $|G^1_N| \times |G^2_N|$, whereas the second one is in space $|G^1_N|$

A first approximation This first algorithm is based on the fact that $S(N,M) \in [0;1]$. We do not calculate the exact value which allows us to compute independantly of the others nodes of G^1 which nodes of G^2 look the more like the nodes of G^1.

We define $S_{approx}(N,M) : G^1_N \times G^2_N \to \mathcal{P}([0;1])$

$$\begin{aligned}S_{approx}(N,M) =& \frac{1}{|\,Vars\,| + 2|\,Fields\,|} \times \\ & \left(\sum_{x \in Vars} \delta_{approx,Vars}(N,M,x) \bigoplus \sum_{f \in Fields} \delta_{approx,in}(N,M,f) \bigoplus \sum_{f \in Fields} \delta_{approx,out}(N,M,f) \right)\end{aligned}$$

where :

$$\begin{aligned}\delta_{approx,Vars} : & G^1_N \times G^2_N \times Vars && \to \mathcal{P}([0;1]) \\ & (N,M,x) && \mapsto \{\neg \left([(S,N,x) \in G^1_E] \ xor \ [(S,M,x) \in G^2_E]\right)\}\end{aligned}$$

where :

$$\begin{aligned}\delta_{approx,in} : & G^1_N \times G^2_N \times Fields && \to \mathcal{P}([0;1]) \\ & \{1\} && \text{if } S_N = \emptyset \wedge S_M = \emptyset \\ & && \text{where } S_N = \{N_0 | (N_0,N,f) \in G^1_E\} \text{ and } S_N = \{M_0 | (M_0,M,f) \in G^1_E\} \\ & \{0\} && \text{if } S_N = \emptyset \ xor \ S_M = \emptyset \\ & [0;1] && \text{otherwise}\end{aligned}$$

and where $\delta_{approx,out} : G^1_N \times G^2_N \times Fields \to [0;1]$ is defined as $\delta_{approx,in}$ where $S_N = \{N_0 | (N,N_0,f) \in G^1_E\}$ and $S_N = \{M_0 | (M,M_0,f) \in G^1_E\}$.

Let us also define $average([a;b]) = \frac{a+b}{2}$.

This allows us to define $nodesApproximation : Graph \times Graph \times float \to \mathcal{P}(Node \times \mathcal{P}(Node))$

```
let $nodesApproximation(G^1, G^2, criteria)$ =
    let $resultNodes = \emptyset$
    for $n_1$ in $G^1_N$ :
        let $corresponding = \Big\{ v \in G^2_N | \forall v_2 \in G^2_N \Big( \big( average(S(n_1,v)) > average(S(n_1,v_2)) \big) \vee \big( average(S_{approx}(n_1,v)) = average(S_{approx}(n_1,v_2)) \wedge max(S_{approx}(n_1,v)) \geq max(S_{approx}(n_1,v_2)) \big) \Big) \wedge \Big( average(S_{approx}(n_1,v)) \geq (1 - criteria) \Big) \Big\}$ in
        $resultNodes$.append(($n_1, corresponding$))
    end for
    return $resultNodes$
```

A costlier and more precise algorithm We define the similitude between N and M by induction :

$$\begin{aligned}S_0(N,M) =& [0;1] \\ S_{i+1}(N,M) =& \frac{1}{|\,Vars\,| + 2|\,Fields\,|} \times \\ & \left(\sum_{x \in Vars} \delta_{Vars}(N,M,x) \bigoplus \sum_{f \in Fields} \delta_{i+1,in}(N,M,f) \bigoplus \sum_{f \in Fields} \delta_{i+1,out}(N,M,f) \right)\end{aligned}$$

where :

$$\begin{aligned}\delta_{Vars} : & G^1_N \times G^2_N \times Vars && \to \mathcal{P}([0;1]) \\ & (N,M,x) && \mapsto \neg \left\{ \left([(S,N,x) \in G^1_E] \ xor \ [(S,M,x) \in G^2_E]\right) \right\}\end{aligned}$$

where :

$$\begin{array}{lll} \delta_{i+1,in} : G^1_N \times G^2_N \times Fields & \to \mathcal{P}([0;1]) & \\ (N, M, f) & \mapsto \dfrac{1}{|S_N| \times |S_M|} \displaystyle\sum_{N_0 \in S_N, M_0 \in S_M} S_i(N_0, M_0) & \\ & \text{where } S_N = \{N_0 | (N_0, N, f) \in G^1_E\} \text{ and } S_N = \{M_0 | (M_0, M, f) \in G^1_E\} & \\ 1 & \text{if } S_N = \emptyset \wedge S_M = \emptyset & \\ 0 & \text{if } S_N = \emptyset \ xor \ S_M = \emptyset & \end{array}$$

and where :

$$\begin{array}{lll} \delta_{i+1,out} : G^1_N \times G^2_N \times Fields & \to \mathcal{P}([0;1]) & \\ (N, M, f) & \mapsto \dfrac{1}{|S_N| \times |S_M|} \displaystyle\sum_{N_0 \in S_N, M_0 \in S_M} S_i(N_0, M_0) & \\ & \text{where } S_N = \{N_0 | (N, N_0, f) \in G^1_E\} \text{ and } S_N = \{M_0 | (M, M_0, f) \in G^1_E\} & \\ 1 & \text{if } S_N = \emptyset \wedge S_M = \emptyset & \\ 0 & \text{if } S_N = \emptyset \ xor \ S_M = \emptyset & \end{array}$$

Let us also define $gap([a;b]) = (b - a)$.

Please note that the previous algorithm *nodesApproximation* corresponds to *nodesIncremental*, except that we restrict ourselves to one single step. This allows us to define $nodesIncremental : Graph \times Graph \times float \to \mathcal{P}(Node \times \mathcal{P}(Node)) \times (\mathcal{P}([0;1])matrix)$

```
let $nodesIncremental(G^1, G^2, criteria)$ =
    let $table =$ matrix of size $|G^1_N| \times |G^2_N|$ where $\forall j,k \ table_{j,k} = [0;1] = S_O(n_j, m_k)$ in
    let $prevTable =$ matrix of size $|G^1_N| \times |G^2_N|$ where $\forall j,k \ table_{j,k} = [0;1] = S_O(n_j, m_k)$ in
    We can write $G^1_N = \{n_1, ..., n_{|G^1_N|}\}$ and $G^2_N = \{m_1, ..., m_{|G^2_N|}\}$.
    let $i = 0$
    while $max\{gap(table_{j,k}) | j \in [\![0; |G^1_N|]\!] \wedge k \in [\![0; |G^2_N|]\!]\} > criteria$ do
        let $prevTable = table$ in
        let $i = i + 1$ in
        (*$prevTable$ represents $S_i$ and we are now computing $S_{i+1}$*)
        $\forall j,k \ table_{j,k} = S_{i+1}(n_j, m_k)$ where $S_i = prevTable$
    end while

    let $resultNodes = \emptyset$
    for $n_j$ in $G^1_N$ :
        $corresponding = \Big\{ m \in G^2_N | \forall v \in G^2_N \Big( \Big( average(table_{n,m}) > average(table_{n,v}) \Big) \vee \Big( average(table_{n,m}) = average(table_{n,v}) \wedge max(table_{n,m}) \geq max(table_{n,v}) \Big) \Big) \wedge \Big( average(table_{n,m})) \geq (1 - criteria) \Big) \Big\}$ in
        $resultNodes$.append(($n_1, corresponding$))
    end for
    return ($resultNodes, table$)
```

Union of the graphs From now, we are going to use *nodesApproximation*, it can be replaced with small modifications by *nodesIncremental*. These modifications will be marked as commentaries.

We are first going to define *unionNodes*: its aim is to give the sets of nodes that are considered equivalent. For each node n_j in the first graph, we construct two sequences of nodes that are considered close. We create two different sequences in order to differenciate the nodes of G^1 and those of G^2. We stop the construction when either we reach a fixpoint or when there exists two nodes that can no longer be considered close.

$unionNodes : Graph \times Graph \times float \to \mathcal{P}(\mathcal{P}(Node) \times \mathcal{P}(Node))$

```
let $unionNodes(G^1, G^2, criteria)$ =
    let $resultNodes^1 = \emptyset$ in
    let $resultNodes^2 = \emptyset$ in
    let $prevResultNodes^1 = \emptyset$ in
    let $prevResultNodes^2 = \emptyset$ in
    let $EnsNodes = \emptyset$ in
    We can write $G^1_N = \{n_1, ..., n_{|G^1_N|}\}$ and $G^2_N = \{m_1, ..., m_{|G^2_N|}\}$.
    let $nodesToTreat^1 = G^1_N$ in
    let $nodesToTreat^2 = G^2_N$ in
    let $nodesSimilarity = nodesApproximation(G^1, G^2, criteria)$
    (*let $(nodesSimilarity, tableSimilarity) = nodesApproximation(G^1, G^2, criteria)$*)
    for $n_j$ in $nodesToTreat^1$ :
        let $correspondingNodes$ such that $(n_j, correspondigNodes) \in nodesSimilarity$
        $resultNodes^1 = \{n_j\}$
        $resultNodes^2 = M$ where $(n_j, M) \in nodesSimilarity$
        $prevResultNodes^1 = prevResultNodes^2 = \emptyset$
        while $\neg\Big((resultNodes^1 \subset prevResultNodes^1 \wedge resultNodes^2 \subset prevResultNodes^2) \vee$
        $(max\{gap(S_{approx}(n_i, m_k))|n_i \in resultNodes^1 \wedge m_k \in resultNodes^2\} < (1 - criteria))\Big)$
        (*$(max\{gap(tableSimilarity_{i,k})|n_i \in resultNodes^1 \wedge m_k \in resultNodes^2\} < (1 - criteria))\Big)$*)
            (*iteration step*)
            $prevResultNodes^1 = resultNodes^1$
            $prevResultNodes^2 = resultNodes^2$
            $resultNodes^2 = \{m_k|\exists n_i \in prevResultNodes^1\ \ \exists A\ \ m_k \in A \wedge (n_i, A) \in nodesSimilarity\}$
            $resultNodes^1 = \{n_i|\exists m_k \in prevResultNodes^2\ \ \exists A\ \ m_k \in A \wedge (n_i, A) \in nodesSimilarity\}$
        end while

        if $\neg(max\{gap(S_{approx}(n_i, m_k))|n_i \in resultNodes^1 \wedge m_k \in resultNodes^2\} < (1 - criteria))$ then
        (*if $\neg(max\{gap(tableSimilarity_{i,k})|n_i \in resultNodes^1 \wedge m_k \in resultNodes^2\} < (1 - criteria))$
then*)
            (*case where the criteria is respected : a fixpoint has been reached*)
            $nodesToTreat^1$.remove($resultNodes^1$)
            $nodesToTreat^2$.remove($resultNodes^2$)
            $EnsNodes$.append(($resultNodes^1, resultNodes^2$))
        else
            (*the criteria is not respected*)
            $nodesToTreat^1$.remove($n_j$)
            $EnsNodes$.append(($\{n_j\}, \emptyset$))
        end if
    end for
    for $m_k$ in $nodesToTreat^2$ :
        $EnsNodes$.append(($\emptyset, m_k$))
    end for
    return $EnsNodes$
```

We then define $lowerCardinality$ which gives the lower bound of $\mathcal{C}$ containing a certain cardinality. Let us first recall the definition of the distance $d(u, v)$ with u, v in $\mathcal{P}(\mathbb{N})$: $d(u, v) = |\{x \in u \setminus v\} \cup \{y \in v \setminus u\}|$.

We also define $d_{include,min}(c) = min\{d(c, card)|card \in \mathcal{C} \wedge c \subset card\}$.

This allows us to define : $lowerCardinality(c) = card$ where $\forall card_1 \in possibilitySet$ $card$ is lower than $card_1$ (which means that the last cardinality has to bear the worst imprecison, then the previous one etc) with $possibilitySet = \{card \in \mathcal{C}|c \subset card\ \wedge\ d_{include,min}(c) = d(c, card)\}$

We are able to define a lattice on the set of properties. Let us first define the order on properties :
let p and q two properties $(\mathcal{P}(Node) \to \mathcal{P}(MS))$, we say that $(p \preceq q)$ when for all N in the domain of p and for all M is in the domain of q and $p(N) \subset q(M)$.
We can define $\top : \mathcal{P}(Node) \to \{MS\}$ and $\bot$ which is defined nowhere. This gives us a structure of lattice on the properties. Looking for the least upper bound of p and q can be done on this lattice by the operation $\sqcup$.
For N in G_N, if N is not in the domain of G_π, then we consider it as being $\top$.

Given $unionNodes$, we can define $union : Graph \times Graph \times float \to Graph$

```
let $union(G^1, G^2, criteria)$ =
    let $EnsNodes = unionNodes(G^1, G^2, criteria)$ in
    We can write $EnsNodes = \{(N_1, M_1), ..., (N_a, M_a)\}$
    We can write $G_N^1 = \{n_1, ..., n_{|G_N^1|}\}$ and $G_N^2 = \{m_1, ..., m_{|G_N^2|}\}$.
    let $G_N^{sum} = \{v_1, ..., v_a\}$ where $v_1, ..., v_a$ are new nodes in
    let $G_E^{sum} = \{(v_i, v_j, f) | \exists n_{init} \in N_i\ \exists n_{end} \in N_j\ (n_{init}, n_{end}, f) \in G_E^1\} \cup \{(v_i, v_j, f) | \exists m_{init} \in M_i\ \exists m_{end} \in M_j\ (m_{init}, m_{end}, f) \in G_E^2\}$ in
    let $c_{G^{sum}} : Node \cup Edge \to \mathcal{P}(\mathbb{N})$
    $\begin{cases} c_{G^{sum}}(v_i) = lowerCardinality\left(\bigcup_{n \in N_i} c_{G^1}(n) \ \cup \ \bigcup_{m \in M_i} c_{G^2}(m)\right) \text{ where } v \in Node \\ c_{G^{sum}}(e = (v_i, v_j, f)) = lowerCardinality\left(\bigcup_{\substack{n^1 \in N_i \\ n^2 \in N_j}} c_{G^1}(n^1, n^2, f) \ \cup \ \bigcup_{\substack{m^1 \in M_i \\ m^2 \in M_j}} c_{G^2}(m^1, m^2, f)\right) \\ \text{ where } e \in Edge \text{ and where } C_{G^z}(e) = \{0\} \text{ if } e \notin G_E^z \end{cases}$
    let $G_\pi^{sum}(V) = \bigsqcup_{v_i \in V} (G_\pi^1(N_i) \sqcup G_\pi^2(M_i))$ when $\forall v_i \in V\ N_i$ in the domain of $G_\pi^1$ and $M_i$ in the domain of $G_\pi^2$ in
    return $(G_N^{sum}, G_E^{sum}, G_\pi^{sum}, c_{G^sum})$
```

3 Proofs of correction

3.1 Abstract Interpetation

Let us define the concrete semantics

Definition 2. *We give ourselves a set of memory locations Loc , a set of names $Fields$, a set of variable names $Vars$ and a set of basic values Val . We also set $Label = Vars \cup Fields$. The variables defined in the program are modeled with a stack memory location with fields the variable names, in the set $Vars$.*

$$\begin{aligned} stack : &\quad Vars \rightharpoonup Loc \\ \sigma : &\quad Loc \to Ob \cup \{\textbf{\textit{null}}\} \cup Val \end{aligned}$$

Where we define $o \in Ob$ as $Fields \rightharpoonup Loc$

We also introduce memory states :

Definition 3. *a memory state is a function that maps each couple of memory location and field name to either a memory state, a basic value or the **null** value. We define $MS = Loc \times Fields \to Loc \cup Val \cup \textbf{\textit{null}}$ the set of all memory states.*

We can link the concept of memory state $m \in MS$ to the concrete heap previously defined by

$$\begin{cases} \forall l_1 \in Loc\ \forall l_2 \in Loc \cup Val\ \forall f \in Fields\ (m(l_1, f) = l_2 \iff \exists o \in Ob\, \sigma(l_1) = o \ \wedge \ o(f) = l_2) \\ \forall x \in Vars\ (m(stack, x) = Env(l)) \end{cases}$$

Definition 4. *Let us define the abstraction considered.*

$$\begin{tikzcd} B & A \arrow[l, bend right, "\gamma"'] \end{tikzcd}$$

where B is the concrete semantics : $B \in \mathcal{P}(MS)$, A is the abstract semantics, in our case, a shape graph and $\gamma : G \to \{m | m \,\tilde{\in}\, G\}$

We also allow the notation of a memory state to be extended : we define by induction $\forall m \in MS\ \forall l \in Loc\, \forall p \in Fields^*\ \forall f \in Fields$

$$\begin{cases} m(l, p.f) = m(m(l, p), f) \text{ if } m(l, p) \in Loc \text{ and } m(m(l, p), f) \in Loc \\ m(l, \epsilon) = l \end{cases}$$

As we are going to manipulate many graphs at the same time in this section, we may add in parentheses the name of the considered graph (ex : $stack_p(G)$) when this is necessary for comprehension. You should also pay attention to the fact that in $m \,\tilde{\in}_\phi\, G$, the choice of ϕ does not affect the nodes, edges, properties of G. Therefore, we can select any ϕ.

3.2 focus

In order to show that focus is correctly defined, we have to show that the abstraction obtained is contained in the previous abstraction :

Property 1. *let G be a shape graph and let p be a path in $Fields^*$.*

$$\gamma(G) \subseteq \bigcup_{G' \in focus(G,P)} \gamma(G')$$

Lemma 1. *let G be a shape graph, let p be a path in $Fields^*$ and let f be a $Fields$.*

$$G' \in focus(G,p) \Rightarrow focus(G',f) \subset focus(G,p.f)$$

Lemma 2. *let i an integer and let $d_0, ..., d_k$ k integers such that $\sum_{a=0}^{k} d_a = i$, then :*

$$\forall t \in [\![1; i]\!] \; \exists ! j \text{ such that } t \in \left[\!\!\left[1 + \sum_{a=0}^{j-1} d_a ; \sum_{a=0}^{j} d_a \right]\!\!\right]$$

Proof. Let us first proove the existence of such j :
for $j = 0$ we have $1 + \sum_{a=0}^{j-1} d_a = 1$, and for $j = k$ we get $\sum_{a=0}^{k} d_a = i$.
Moreover, $\sum_{a=0}^{j} d_a = \left(1 + \sum_{a=0}^{j} d_a\right) - 1$, therefore, there always exist such an interval as $t \in \mathbb{N}$.

Let us now prove the unicity : let us assume there are two such j-s : j and $j + h$.

$$\begin{aligned} & t \leq \sum_{a=0}^{j} d_a \text{ by definition of } j + h \\ & < 1 + \sum_{a=0}^{j} d_a \leq 1 + \sum_{a=0}^{j} d_a + \sum_{b=0}^{h-1} d_{j+1+h} = 1 + \sum_{a=0}^{h-1} d_a \\ & \leq t \text{ by definition of } j + h \end{aligned}$$

Therefore there are no such two j. □

Lemma 3. *let m be a memory state, let G be a shape graph and let p be a path in $Fields^*$.*

$$m \,\tilde{\in}\, G \Rightarrow \exists G' \in focus(G,p) \text{ such that } m \,\tilde{\in}\, G'$$

Proof. We are going to show this result by induction on p.
base case : if $p = \epsilon$ then $focus(G,p) = G$
If we have the result for p, let $G' \in focus(G,p)$ such that $m \,\tilde{\in}\, G'$. Let us now show : $\exists G'' \in focus(G',f)$ such that $m \,\tilde{\in}\, G''$.
By induction $m \,\tilde{\in}\, G'$ therefore we have $\phi : loc \to G'_N$ such that :

- for all (l,q) in the domain of m, if $m(l,q) \in Loc$ then $(\phi(l), \phi(m(l,q)), q) \in G'_E$
- for all $N \subset G'_N$ in the domain of G'_π, m restricted to $\phi^{-1}(N)$ is in $G'_\pi(N)$

which we will note $m \,\tilde{\in}_\phi\, G'$

Let us now show that there exists a $\psi : Loc \to G''_N$ such that $m \,\tilde{\in}_\psi\, G''$. There are only two possible cases for the value of $\phi(l)$: it can either be in $stack_{p.f}$ or not.

- $\forall n \notin stack_{p.f}$ $n \in G'_N \iff n \in G''_N$ therefore, we define $\phi(l) = \psi(l)$ for all l in Loc such that $\phi(l) = n$
- otherwise, let us note $(k, \{c_0, ..., c_k\}) = cardinalityNewNode(G', n)$ and only three cases are possible :
  - $k = 0$ in which case $c_{G'}(n) = 1$ so there is only one $l \in Loc$ such that $\phi(l) = n$. We define $\psi(l) = n_0$
  - if we have found the corresponding k and $c_0, ..., c_k$:
    Let $\{l_1, ..., l_i\} = \phi^{-1}(\{n\})$.
    We have $i \in \bigoplus_{j=0}^{k} c_j$. Therefore there exists $(d_0, ..., d_k) \in c_0 \times ... \times c_k$ such that $\sum_{j=0}^{k} d_j = i$.
    Let us first define $\psi(l_1), ..., \psi(l_i)$, in order to have all edges corresponding to $m(l_b, label) = l_c$ in at least one graph.

- $*$ If no such $\{l_1, ..., l_i\}$ exists then there is no location corresponding to n. As $\phi^{-1}(\{n\}) = \emptyset$, we had $0 \in c_{G'}(n)$, therefore, there exists a graph G'' obtained through $focusNodes$such that n no longer exists in it.
- $*$ If n loops on itself with an f-labbeled edge in G' and n is reachable through p, then without loss of generality we may assume $m(l_1, f) = l_2$ or $m(l_1, f) = l_1$. We define $\forall j \in [\![0; k]\!]$ $\forall t \in [\![1 + \sum_{a=0}^{j-1} d_a; \sum_{a=0}^{j} d_a]\!] \psi(l_t) = node_{n,j}$. This is possible thanks to the lemma 2.
  There is exactly one graph G'' such that the edge $(\psi(l_1), \psi(l_1), f) = (node_{n,0}, node_{n,0}, f)$ exists in it. Similarly for the edge $(\psi(l_1), \psi(l_2), f) = (node_{n,0}, node_{n,1}, f)$ exists. All other edges corresponding to $label = f$, $b \in [\![2; i]\!]$ and $c \in [\![1; i]\!]$ exists in all the graphs that are created.
- $*$ If $m(l_a, p) = l_b$ where $\phi(l_a) = S$ and $label = f$ and $l_c \notin \{l_1, ..., l_i\}$ then we may assume without loss of generality that $b = 1$. The edge $(\phi(l_b), \phi(l_c), f)$ existed in G' therefore, by setting $\forall j \in [\![0; k]\!]$ $\forall t \in [\![1 + \sum_{a=0}^{j-1} d_a; \sum_{a=0}^{j} d_a]\!] \psi(l_t) = node_{n,j}$, we have that there exists G'' such that the edge $(\psi(l_b), \phi(l_c), f)$is in G''_E.
- $*$ In all other cases, the corresponding edge exists in all graphs obtained by $focusEdges$

This proves the first point of the embedding.

Let us now prove the second point. Let N a subset of G''_N in the domain of G''_π where G'' has previously been selected. G''_π is modified in the $propUpdate$ method called on G''

- if $n \in N \Rightarrow n \in G'_N$, then $\phi^{-1}(N) = \psi^{-1}(N)$ and $G'_\pi(N) = G''_\pi(N)$, therefore the property is verified.
- if $N = \{m\}$ such that m was one of the nodes added to G'' : $\exists n\ \exists A\ (n, A) \in nodesAdded(G')\ \wedge m \in A\ \wedge n \in P$ where $P \subset G'_N$ with $intensive(prevProp|_P)$. Note that we may have $P = \{n\}$. We have by definition of ψ that $\phi^{-1}(\{n\}) = \psi^{-1}(\{m\})$, and by definition of $propUpdate$: $G''_\pi(\{m\}) = G'_\pi(P)$.
  m restricted to $\phi^{-1}(P)$ is in $G'_\pi(N)$ then, thanks to the distributivity of the function and as $n \in P$, m restricted to $\psi^{-1}(\{m\}) isin G''_\pi(\{m\})$
- otherwise : $\exists P \subset G'_N$ in the domain of G'_π such that $P = N \cup \{n | \exists A\ (n, A) \in nodesAdded(G')\} \setminus \{m | \exists n\ \exists A\ (n, A) \in nodesAdded(G')\ \wedge m \in A\}$ and $\neg intensive(prevProp|_P)$.
  m restricted to $\phi^{-1}(P)$ is in $G'_\pi(P)$. According to the definition of $propUpdate$, $G'_\pi(P) = G''_\pi(N)$ and according to the definition of ψ, $\psi^{-1}(N) = \phi^{-1}(P)$. Therefore, m restricted to $\psi^{-1}(N)$ is in $G''_\pi(N)$

We have shown that : $\exists G'' \in focus(G', f)$ such that $m \,\tilde{\in}_\psi\, G''$ and as $focus(G', f) \subset focus(G, p.f)$: $\exists G'' \in focus(G, p.f)$ such that $m \,\tilde{\in}\, G''$. □

Given this property, we will now show the property 1

Proof. let $m \,\tilde{\in}\, G$.
$\Rightarrow \exists G' \in focus(G, p). m \,\tilde{\in}\, G'$
$\Rightarrow m \in \bigcup_{G' \in focus(G,p)} \gamma(G')$

□

3.3 assign

Let us define the effect of an assignment on the memory states :

Definition 5. *let m be a memory state before the assignment, let m' the memory state obtained after performing the assignment $p.f = r$ where $p \in Label^*$ and $f \in Label$. We define l_1 as the location such that $m(Stack, p) = l_1$, then we have two different cases :*

- *if $r \in Vars\,.\,Fields^*$, $m(Stack, r) = l_2$ m' :* $\begin{cases} m'(l, q) = m(l, q) & \text{for } l \neq l_1 \ \vee\ q \neq f \\ m'(l_1, f) = l_2 \end{cases}$
- *if $r \in Val$, m' :* $\begin{cases} m'(l, q) = m(l, q) & \text{for } l \neq l_1 \ \vee\ q \neq f \\ m'(l_1, f) = r \end{cases}$

let us note $assignment(p.f = r, m)$ such m'

we can now formally define the method on G $values : Node \to Val$ by

$$\forall n \in G_N \; G.values(n) = \{\sigma(l) \notin Loc | m \,\tilde{\in}_\phi\, G \;\wedge\; l \in \phi^{-1}(n)\}$$

In order to show that assign is correctly defined , we have to show that the abstraction obtained is contained in the previous abstraction :

Property 2. *let G be a shape graph, let $p \in Label^*$, let $f \in Label$ and let $r \in Label \cup Val$.*

$$\{assignment(p.f = r, m) | m \in \gamma(G)\} \subseteq \bigcup_{G' \in assign(G, p.f = r)} \gamma(G')$$

Lemma 4. *let G be a shape graph, let $p \in Label^*$, let $f \in Label$ and let $r \in Label$*

$$\{assignment(p.f = r, m) | m \in \gamma(G)\} \subseteq \bigcup_{G' \in assignLoc(G, p.f = r)} \gamma(G')$$

Proof. Let $m \in G : m \,\tilde{\in}\, G$. Let $m' = assignment(p.f = r, m)$, we have $\begin{cases} m'(l,q) = m(l,q) \text{ if } (l,q) \neq (l_1, f) \\ m'(l_1, f) = l_2 \end{cases}$
where $m(stack, p) = l_1$ and $m(stack, r) = l_2$.
As $m \,\tilde{\in}\, G$, by the lemma 3 $\exists G'' \in focus(focus(G, p), r) \; m \,\tilde{\in}_\phi\, G$. Let G' be the graph obtained after performing $assignLoc$on G'' (which is similar to only performing the loop on G''). Let us now prove that we also have $m' \,\tilde{\in}_\phi\, G'$:

- for all (l,q) in the domain of m, $m(l,q) \in Loc \Rightarrow (\phi(l), \phi(m(l,q)), q) \in G'_E$. Let (l,q) in the domain of m' such that $m'(l,q) \in Loc$:
  - either $(l,q) \neq (l_1, f)$, we then have $m'(l,q) = m(l,q)$ therefore $(\phi(l), \phi(m'(l,q)), q) = (\phi(l), \phi(m(l,q)), q) \in G''_E$. Moreover, $G'_E = G''_E \backslash \{(n_p, n, f) \in G''_E | n_p$ reachable through $p\} \cup \{(n_p, n_r, f) | n_p$ reachable through $p \wedge n_r$ reachable through $r\}$. Therefore if $q \neq f$, $(\phi(l), \phi(m'(l,q)), q) \in G'_E$. If $q = f$, as $l \neq l_1$, we also have $(\phi(l), \phi(m'(l,q)), q) \in G'_E$.
  - or $(l,q) = (l_1, f)$: let us note $n_p = \phi(l_1)$ and $n_r = \phi(l_2)$. $(\phi(l_1), \phi(m'(l_1, f)), f) = (\phi(l_1), \phi(l_2), f) = (n_p, n_r, f) \in G'_E$.
- for all $N \subset G_N$ in the domain of G_π, m restricted to $\phi^{-1}(N)$ is in $G_\pi(N)$. As $G'_N = G_N$, $G'_\pi = G_\pi$, and given the definition of m', we have : for all $N \subset G'_N$ in the domain of G'_π, m' restricted to $\phi^{-1}(N)$ is in $G'_\pi(N)$.

this proves that $m' \,\tilde{\in}\, G'$ □

Lemma 5. *let G be a shape graph, let $p \in Label^*$, let $f \in Label$ and let $r \in Val$*

$$\{assignment(p.f = r, m) | m \in \gamma(G)\} \subseteq \bigcup_{G' \in assignVal(G, p.f = r)} \gamma(G')$$

Proof. Let $m \in G : m \,\tilde{\in}\, G$. Let $m' = assignment(p.f = r, m)$, we have $\begin{cases} m'(l,q) = m(l,q) \text{ if } (l,q) \neq (l_1, f) \\ m'(l_1, f) = r \end{cases}$
where $m(stack, p) = l_1$
As $m \,\tilde{\in}\, G$, by the lemma 3 $\exists G'' \in focus(focus(G, p), r) \; m \,\tilde{\in}_\phi\, G$. Let G' be the graph obtained after performing $assignVal$on G'' (which is similar to only performing the loop on G''). Let us now prove that we also have $m' \,\tilde{\in}_\phi\, G'$:

- let (l,q) in the domain of m' such that $m'(l,q) \in Loc$. As $m'(l_1, f) = r \in Val$, we have $(l,q) \neq (l_1, f)$ which means that $m'(l,q) = m(l,q)$. Therefore $(\phi(l), \phi(m(l,q)), q) \in G''_E$. As $G'_E = G''_E \setminus \{(n_p, n, f) \in G''_E | n_p$ reachable through $p\} \cup \{(n_p, n_r, f) | n_p$ reachable through $p \;\wedge\; n_r$ of value $r\}$ and as $\phi(l_1) = n_p$ and $(l,q) \neq (l_1, f)$, $(\phi(l), \phi(m(l,q)), q) \in G'_E$.
- let $N \subset G'_N$. If $N \subset G''_N$, as $m \,\tilde{\in}\, G''$ we have the similar property for m' and G'. Otherwise, $G'_N = G''_N \cup \{n_r\}$ with $G'.values(n_r) = r$. As $\phi^{-1}(\{n_r\}) = \emptyset$, we also have that m' restricted to $\phi^{-1}(N)$ is in $G_\pi(N)$

this proves that $m' \,\tilde{\in}\, G'$ □

We may now prove the property 2 of the correctness of assign.

Proof. The proof of property 2 is straightforward given the lemmas 4 and 5 □

3.4 Proving union

Property 3. *Given $m \in MS$, G^1 and G^2 two graphs and criteria $\in [0;1]$ we have :*

$$\begin{cases} m \mathrel{\tilde{\in}} G^1 \Rightarrow m \mathrel{\tilde{\in}} union(G^1, G^2, criteria) \\ m \mathrel{\tilde{\in}} G^2 \Rightarrow m \mathrel{\tilde{\in}} union(G^1, G^2, criteria) \end{cases}$$

Lemma 6. *Given G^1 and G^2 two graphs and criteria $\in [0;1]$, we note $\{(N_1, M_1), ..., (N_a, M_a)\} = unionNodes(G^1, G^2, criteria)$. Then :*

$$\begin{cases} \forall i, j \ N_i \cap N_j = \emptyset \text{ and } \bigcup_{j=0}^{a} N_j = G^1_N \\ \forall i, j \ M_i \cap M_j = \emptyset \text{ and } \bigcup_{j=0}^{a} M_j = G^2_N \end{cases}$$

Proof. We can restrict the proof to only the case with G^1 as the other one is exactly the same. We note $\{(N_1, M_1), ..., (N_a, M_a)\} = unionNodes(G^1, G^2, criteria)$. We note $G^{sum} = union(G^1, G^2, criteria)$

let $m \mathrel{\tilde{\in}}_\phi G^1$. let us define ψ : for l in the domain of ϕ such that $\phi(l) = n \in N_i$ (we are able to choose such unique i thanks to the lemma 6), we choose $\psi(l) = v_i$.

- let $l_1, l_2 \in Loc$, let $f \in Fields \cup Vars$ such that $m(l_1, f) = l_2$.
  Let $n^1 = \phi(l_1) \ \in N_i$, let $n^2 = \phi(l_2) \ \in N_j$.
  By hypothesis, $m \mathrel{\tilde{\in}}_\phi G^1$, therefore $(\phi(l_1), \phi(l_2), f) \in G^1_E$. Moreover, by definition of ψ, we have $\psi(l_1) = v_i$ and $\psi(l_2) = v_j$.
  By construction of G^{sum}_E, $(\psi(l_1), \psi(l_2), f) \in G^{sum}_E$.

- let V in the domain of G^{sum}_π.
  By construction of G^{sum}_π, for all $v_i \in V$ there exist N_i in the domain of G^1_π and M_i in the domain of G^2_π such that $G^{sum}_\pi(V) = \bigsqcup_{v_i \in V}(G^1_\pi(N_i) \sqcup G^2_\pi(M_i))$.
  for all v_i, for all l in the domain of ϕ such that $\phi(l) \in N_i$ then $\psi(l) = v_i$ (by construction of ψ). Therefore $\psi^{-1}(V) = \psi^{-1}\left(\bigcup_{v_i \in V} v_i\right) = \bigcup_{v_i \in V} \psi^{-1}(\{v_i\}) = \bigcup_{v_i \in V} \phi^{-1}(N_i)$
  for all $v_i \in V$, N_i is in the domain of G^1_π and $m \mathrel{\tilde{\in}}_\phi G^1$, therefore m restricted to $\phi^{-1}(N_i)$ is in $G^1_\pi(N_i)$. Therefore m restricted to $\bigcup_{v_i \in V} \phi^{-1}(N_i)$ is in $\bigsqcup_{v_i \in V} G^1_\pi(N_i)$ which is included in $\bigsqcup_{v_i \in V}(G^1_\pi(N_i) \sqcup G^2_\pi(M_i)) = G^{sum}_\pi(V)$.

We have proved that $m \mathrel{\tilde{\in}} G^{sum}$ □

4 Conclusion

4.1 Synthesis

We have formalized an approach that allows us to get a shape invariant for the execution of a program. This shape invariant may allow us to check if expected properties are verified or not. Moreover, as we abstract the execution of the program when constructing the invariant, we isolate the cases where the execution of the program will not work and will return an error. This should allow us to add some guards before executing the program.

This approach is very interesting compared to others, in particular TVLA : we are able to join graphs or nodes that are looking alike, we do not have to evaluate properties on each node, but we only register where the property is verified, which is a huge gain of complexity. There are some work being done also on the improvement of TVLA [9], which may lead to some improvement of this method, but further tests are needed in order to compare the two.

As the theoretical background is light-weight compared to TVLA, it is visual and intuitive which makes it very easy to understand and to work with. There are also very interesting and complicated approaches which are linked to categories [12], as in [3]. These approaches have not been developped very far yet. On the contrary our approach is very promising and may lead very quickly to a prototype. It should also be a lot more efficient than previously designed approaches as we union the graphs when possible and we can simplify graphs by analyzing which nodes are similar. This allows us to sum up our results without loosing too much information, as we choose how much information may be forgotten.

4.2 Future Work

There is still a lot of work to be done, both on the theory and on the implementation.

The function giving the order on graphs should be formalized very quickly as it is very important for this approach as a whole : when testing a loop of a program, we “execute” this loop on sets of graphs. At first, each graph before the loop is a subgraph of one of the graphs obtained after the loop. But at one point, we reach a fixpoint and the set of graphs obtained after the loop is contained in the set of graphs we had before the loop. This ordering function has not yet been formalized. Another function which may be interesting is widening.

There are some tests to be done in order to verify that we do gain a lot of complexity using this approach and therefore we should quickly implement our algorithms. We did test a few ideas such as the principle behind the union of graph, but there is still a lot of code to be written. Furthermore, some functions such as *cardinalityNewNode*still need some work before they can be implemented. There is of course some work to be done also on parsing the program that we want to analyse.

There should also be some interesting connexion with region logic which would deserve some further work.

4.3 A word on the methodology and personal conclusion

M.Mauborgne has allowed me to dig into any direction I wished so I was able to go through almost every subject I have studied this year (this should be further developped in my oral examination):

- categories : there exists an approach to this problem using categories which (to my opinion) is very theoretical and will not lead to major results soon [3]. But I did use some ideas of the categories though without success.
- model-checking : I tried to apply some methods of model-checking such as [13] in order to union graphs but it did not work for memory locations that were not accessible from the stack and it was barely working when it was accessible.
- graph automata : unfortunately the results on graph automata were too general for us to represent our transfer functions by some transformations on the graphs, the founding idea is expressed in [8] and there are now mainly two approaches [2] and [4].
- partitioning [5] : this would not have given much result as the constraints are so important that two graphs obtained by focusing on a previous one cannot be unioned back afterwards !
- probabilities : it is the idea behind the similitude used for the union
- ...

This internship has been a great and very enriching experience for which I am very grateful. It has been an opportunity to learn new ideas (skeletons are fun [11]) and to work with notions I had studied the previous year in a totally different way.

It has also given me an overview of the way academic research works and I am very eager to start a Ph.D.

References

[1] C. Baier, J.P. Katoen, et al. Principles of model checking. 2008.

[2] A. Bakewell, D. Plump, and C. Runciman. Checking the shape safety of pointer manipulations. *Relational and Kleene-Algebraic Methods in Computer Science*, pages 48–61, 2004.

[3] C. Blume, S. Bruggink, and B. König. Recognizable graph languages for checking invariants. *Electronic Communications of the EASST*, 29(0), 2010.

[4] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking of complex dynamic data structures. *Static Analysis*, pages 52–70, 2006.

[5] A. Cardon and M. Crochemore. Partitioning a graph in o (— a— log2— v—). *Theoretical Computer Science*, 19(1):85–98, 1982.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[7] P. Cousot and R. Cousot. *A gentle introduction to formal verification of computer systems by abstract interpretation*, pages 1–29. NATO Science Series III: Computer and Systems Sciences. IOS Press, 2010.

[8] P. Fradet and D. Le Métayer. Shape types. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 27–39. ACM, 1997.

[9] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. *Static Analysis*, pages 159–412, 2004.

[10] Mark Marron. Structural analysis : Combining shape analysis information with points-to analysis computation. to be published.

[11] L. Mauborgne. An incremental unique representation for regular trees. *Nordic Journal of Computing*, 7(4):290–311, 2000.

[12] P.A. Mellies. Categorical models of linear logic revisited. *Theoretical Computer Science*, 2002.

[13] S.Schwoon P.Gastin. Course of basics of verification. 2010.

[14] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.