Abstract Interpretation Based On Forester Automata

Martin Hruška
ihruska@fit.vutbr.cz

Abstract

Shape analysis is a field of static analysis focusing on verification of programs manipulating dynamic data structures. An approach to shape analysis described in this work is based on combination of abstract interpretation and forest automata. This paper presents abstract interpretation as a general method for defining semantics of programs first. Then forest automata are defined and their ability to model dynamic data structure is presented. Finally, an application of abstract interpretation based on forest automata in shape analysis is covered.

Contents

1 Introduction

Abstract interpretation based on forest automata is a technique of *shape analysis*. Shape analysis is a field of static analysis aiming to formal verification of memory safety properties (safety properties guarantee that something bad never happens) of programs manipulating dynamic data structures. The violations of the safety properties are often bugs such as segmentation fault, invalid frees or memory leaks. Abstract interpretation based on forest automata can detect these violations and provide a counterexample (a particular path in analysed program leading to a memory bug) or it can prove correctness of program (w.r.t. given specification) by generation of shape invariants representing all possible shapes which can be allocated on heap in a given point of analysed program. When we refer to shapes of heap then it means (in context of shape analysis) that the data structures allocated on heap are viewed as graphs forming different shapes.

This work describes both components, abstract interpretation and forest automata, separately and then their synthesis for the purposes of shape analysis is presented.

Abstract interpretation is not only a method of formal verification but it is a general framework for automatic definition or (sound) approximation of program semantics (w.r.t. tracked properties) from its source code. Abstract interpretation does not track the properties of interest in program under its original semantics (in so called *concrete domain*). It represents them in *abstract domain*. This abstract representation is more concise but information-losing. E.g., consider an analysis of possible values of integer variables. The concrete domain is formed by the set of sets of integers and the abstract domain can be set of intervals over integers. In such case, a concrete value $\{2, 10, 100\}$ of a variable could be represented by interval $\langle -100, 100\rangle$ in the abstract domain. It is of course possible to pick a more precise interval such as $\langle 2, 100\rangle$. A choice of interval depends on an abstraction function which maps values from concrete domain to abstract one. An abstract function is often compromise between precision and computational feasibility (i.e., avoiding a state space explosion during analysis). Apart from abstract domain, we also need to define the effects of program statements on observed properties in abstract domain.

A choice of abstract domain and abstraction function is not completely unrestricted. The framework of abstract interpretation requires to chose them to overapproximate all possible behaviors of the original program. Then each possible behavior of the analysed program is included in behaviors of the program in abstract domain. Such overapproximation is *sound* because it will not miss any behavior of the original program. On the other hand, a weakness of this approach is *incompleteness* — occurrence of the behaviors of program not presented in the original semantics.

One can ask for the reasons of the overapproximation when it leads to imprecise information about program semantics. The reasons are two. The first and the main one is that an abstraction (representation in abstract domain) gives analysis chance to terminate since the problem of defining semantics of a program is generally undecidable (by reduction from halting problem). The second reason is an acceleration of computation of semantics by reduction of explored state space in abstract domain. That is in contrast to an analysis in concrete domain which often lead to space explosion problem and so to an exponential complexity.

The information collected by abstract interpretation can be used not only in verification but also in automatic code optimization done by compilers or as a support for debugging (e.g. abstract interpretation can provide data flow graph).

The other essential part of this work is forest automata (FA) theory. Forest automata were introduced in [7]. They are basically tuples of tree automata (TA) (automata similar to finite automata but they accept trees instead of words) which are interconnected by references to another TA of FA in symbols from alphabet of forest automaton. Forest automata have been designed directly for shape analysis and they came from principles of *abstract regular model checking* [2]. However, in [1] they were presented as an abstract domain in abstract interpretation because forest automata can model a certain class of data structure. These data structures can be viewed as graphs (or shapes) that can be decomposed to the tuples of trees (this trees are interconnected by the mentioned references). These tuples of trees can be accepted by forest automata as the members of their languages. Therefore forest automata can represent (by their languages) the possible shapes of heaps and so they can be used as an abstract domain. The abstraction techniques over forest automata were also designed to approximate represented state space and makes possible for analysis to terminate.

Forest automata based abstract interpretation is done by (i) using forest automata as an abstract domain, (ii) definition of the appropriate operations over forest automata modeling statements in program modifying dynamic data structures and (iii) using abstraction over FA to enable approximating all possible shapes of heap in particular program points. The method is able to analyse complex dynamic data structures such as skip-list of the second and the third level which are not automatically analysable by any other technique.

The rest of this work is structured as follows. In Section 2, abstract interpretation will be formally defined and the same is done for forest automata in Section 3. The abstract interpretation based on forest automata is formally described in Section 4.

2 Abstract Interpretation Framework

We already gave some intuition about principles of abstract interpretation in the introduction. This sections provides with more formal and deep description of method. First, the mentioned example about tracking integer variables is completed by other ingredients of abstract interpretation to deepen the intuition. Then the formal definition of abstract interpretation is given, construction of its parts and its properties are described. Finally, a method for computing semantics (or its overapproximation) of program using abstract interpretation is presented.

Consider again the example of the abstract interpretation for tracking the possible values of integer variables. The concrete domain is a set of sets of integers $2^{\mathbb{Z}}$ (for simplification we will not consider overflows of integer variables etc.). The statements of programming language changing values of the variables (e.g., assignment or increment) are called *concrete transformers*. A concrete analysis over the concrete domain with the concrete transformers would led to non-termination of analysis because infinite space may arise. The non-termination problem can be addressed by doing the analysis in abstract domain, which is a set of intervals over $\mathbb{Z}$.

To model the semantics of program in abstract domain, it is necessary to define abstract transformers modeling the concrete semantics. The abstract transformers should reflect an effect of a concrete transformer on the properties tracked by abstract interpretation. In our example, consider the variables x and y with assigned values $(-\infty, -100\rangle$ and $(-100, 100)$ in abstract domain. The abstract transformer corresponding to arithmetic addition $+$ should for $x + y$ give the result $(-\infty, 100)$.

Moreover, we will need to define operator joining possible values of a variable over more paths in program. The examples of program points needing such an operator is the end of `if` statement or the beginning of `while` loop. E.g., the join of a variable x with the abstract value $\langle 100, 200\rangle$ in `then` branch of if statement and $\langle 400, 500\rangle$ in `else` branch would result to the abstract value $\langle 100, 500\rangle$ taking the smaller beginning and greater end of the both intervals as the boundaries of the new interval.

The join operator should result to the greater or equal abstract value compared to its inputs. This is the key property of join operator. An ordering $\sqsubseteq$ over abstract domain is therefore needed for checking of this property of join operator.

Now we will continue from the intuitive description to the formal definition of abstract interpretation (definitions are given in the same way as in [4]).

2.1 Formal Definition of Abstract Interpretation

Abstract Interpretation I of a program P is a tuple $I = (Q, \circ, \sqsubseteq, \top, \bot, \tau)$ where

- Q is an abstract domain
- $\circ : Q \times Q \rightarrow Q$ is a join operator

- $\sqsubseteq \subseteq Q \times Q$ is a relation (ordering) such that $\forall x, y \in Q : x \sqsubseteq y \leftrightarrow x \circ y = y$
- $\top$ is a supremum of Q
- $\bot$ is a infinum of Q
- $\tau = \{Q \times \ldots \times Q \rightarrow Q\}$ is a set of abstract transformers, where a length of the sequence $Q \times \ldots \times Q$ determines arity of abstract transformer

$(Q, \circ, \sqsubseteq, \top, \bot)$ forms a complete semi lattice.

Abstract transformers can be specified by formalization of behavior of concrete instructions of program using e.g., denotational or operational semantics w.r.t. properties tracked by abstract interpretation. Therefore when a programming language has a well-defined semantics then it is easy to use abstract interpretation as a method for computation of a precise program semantics written in this programming language.

2.2 Properties of Abstract Interpretation

The most important characteristics of abstract interpretation are going to be described in the following section. Since abstract interpretation overapproximates possible behaviors of the concrete system it is a **sound** method. A prove of soundness will be informally described using two auxiliary functions α, γ for switching between abstract and concrete domain. An abstraction function $\alpha : P \rightarrow Q$ assigns to an item from concrete domain an item from abstract domain and a concretization function $\gamma : Q \rightarrow P$ maps values from abstract domain back to concrete domain. Assume also an ordering $\leqslant \subseteq P \times P$. Triples $(Q, \sqsubseteq, \alpha)$ and $(P, \leqslant, \gamma)$ form *Gallois connections* [4] which is defined as follows:

$$(Q, \sqsubseteq, \alpha, \gamma, \leqslant, P) \text{ is Gallois connections iff } (Q, \sqsubseteq) \text{ and } (P, \leqslant) \text{ are posets and}$$
$$\forall p \in P\, \forall q \in Q : p \leqslant \gamma(q) \Leftrightarrow \alpha(p) \sqsubseteq q$$

The properties of Gallois connections implies that $\forall p \in P : p \leqslant \gamma(\alpha(p))$. Let $q = \alpha(p)$ then it holds $\alpha(p) \sqsubseteq q$, thus $p \leqslant \gamma(q)$ and finally $p \leqslant \gamma(\alpha(p))$. We get that a concretization of an abstracted value of concrete system is greater than the original value and covers the original value. Another necessary property of abstract interpretation for soundness is that abstract transformers do not violate Gallois connection. Formally, $\alpha(In(p_1, \ldots, p_n)) \leqslant \tau_{In}(\alpha(p_1), \ldots, \alpha(p_n))$ where $p_1, \ldots, p_n \in P$ and τ_{In} is an abstract transformer modeling semantics of an instruction In in concrete program. These two properties of abstract domain and abstract transformers ensure soundness of abstract interpretation.

On the other hand, abstract interpretation is not **complete**. The fact $p \leqslant \gamma(\alpha(p))$ (a concretization of abstracted value is something greater) implies that an abstraction of concrete value may lead to overapproximation of possible behaviors. This is undesirable in the applications such as formal verification where the behaviors added by overapproximation can contain a violation of the checked properties.

A possible solution of problems caused by incompleteness is a refinement of the abstraction. However, an immoderate refinement can led to infeasible computational complexity or even to non-termination of the method. Generally, **termination** of abstract interpretation is guaranteed when abstract domain Q satisfies that Kleene's sequence is finite [4]. This is achieved when abstract domain is finite, or a length of any strictly increasing chain is finite, or abstract domain satisfies the ascending chain condition (every strictly increasing chain is finite, although not bounded). Then the termination comes from Tarski's fixpoint theorem [5]. When abstract domain contains infinite Kleene's sequence then the abstract transformers can be defined to allow only finite Kleene's sequences during abstraction. When the abstract domain has infinite Kleene's sequences and the abstract transformers do not exclude them from computation during abstract interpretation then the sequences should be truncated or computation of fixpoint approximated by some heuristic. Finally, when the infinite sequence are excluded then termination is again guaranteed by Tarski's fixpoint theorem [4].

The abstraction brings inherently certain trade-off between precision and efficiency of the method. When the abstraction is very precise then the analysis is slower because larger state space has to be explored.

2.3 Widening and Narrowing

In abstract interpretation, the mentioned approximation of fixpoint is used in computation of all possible behaviors of a given program point. The approximation is primarily necessary in the loop points of program where a new possible behavior of a program can be generated in each loop cycle. This could lead to unbounded sequence of possible behaviors and nontermination of the analysis. A concept of *widening* is introduced to resolve such cases. The binary operation $\triangledown : Q \times Q \rightarrow Q$ is widening iff:

- $\forall q, q' \in Q : q \circ q' \sqsubseteq q \triangledown q'$
- Every infinite sequence $s_0, \dots, s_i, \dots$ such that $s_0 = C_0$ and $s_i = s_{i-1} \triangledown C_i$, where $C_0, C_i \in Q$, is not strictly increasing.

Informally, using widening $\triangledown$ instead of join $\circ$ is one of possible heuristic which will approximate fixpoint of infinite Kleene's sequence. It also accelerates the fixpoint computation compared with join because $\forall q, q' \in Q : q \circ q' \sqsubseteq q \triangledown q'$.

The opposite operation to widening is *narrowing.* While widening accelerates computation narrowing refines the overapproximation to get a more precise information. Formally, the binary operation $\triangle : Q \times Q \rightarrow Q$ is widening iff:

- $\forall q, q' \in Q : (q' \sqsubseteq q) \Rightarrow q' \sqsubseteq q \triangle q' \sqsubseteq q$
- Every infinite sequence $s_0, \dots, s_i, \dots$ such that $s_0 = C_0$ and $s_i = s_{i-1} \triangle C_i$, where $C_0, C_i \in Q$, is not strictly decreasing.

Although the widening refines the abstraction it does not break conditions guaranteeing fixpoint computation or approximation what comes from the second point of widening definition.

2.4 Computing Semantics Using Abstract Interpretation

We defined abstract interpretation and its basic properties but we have not yet mentioned how to use it for computing semantics of program. Consider a program P with program points denoted P_i where i is an index of a program location. We set the initial values of the tracked semantics properties to $\perp$ for each program point i. Then the corresponding abstract transformers in the given program point are applied repetitively until a fixpoint in each location is reached. The changes in tracked values in one program locations are propagated to each iteration to the subsequent locations. Join, widening (and if needed also narrowing) are applied at the program points such as loop points or ends of conditions where more paths meet. The abstract transformers are applied using round robin algorithm where the order of transformers is defined by the original program locations.

A computation of semantics can be also viewed as a computation of a solution for a system of equitation [3]. Each program location with its dedicated abstract transformer can be viewed as one equitation. The different equations share variables what leads to propagation of values between equitation. An existence of (approximate) solution of this system of equations is guaranteed by existence of a fixpoint in abstract interpretation.

While the first mentioned approach to computing the semantics of a program is closer to implementation perspective, the second one is often used in papers focusing on theoretical aspects of abstract interpretation and the related mathematical background.

Example 2.1 *The working example about tracking integer values will be completed in the notion of the defined framework. The example is taken from [4]. We want to instantiate abstract interpretation framework for approximating possible values of integer variables in given program location. Concrete domain is $2^{\mathbb{Z}}$ and a variable can have one of values from a set of $2^{\mathbb{Z}}$ assigned to it by concrete domain.*

The abstract interpretation is $(I, \circ_I, \langle -\infty, \infty \rangle, \langle , \rangle, \tau)$ where

- $I = \{\langle x, y \rangle \mid x, y \in \mathbb{Z} \wedge x \leqslant y\}$
- $\langle x, x' \rangle, \langle y, y' \rangle \in I : \langle x, x' \rangle \circ_I \langle y, y' \rangle = \langle min(x, y), max(x', y') \rangle$. *Informally, $\circ_I$ is a union of two intervals*
- $\langle , \rangle$ *is an empty interval*
- τ *will be defined on demand for instructions used in the following example program*

Consider the following program in the C language, where C_i in a comment is labelling of a particular program point i.

Listing 1: Example program

```
void main()
{
        // C_0
        int x = 0;
        // C_1

        // C_2 -- join point of the cycle
        while (x <= 100)
        {
                // C_3
                x = x+1;
                // C_4
        }
        // C_5
}
```

The only integer variable in program is x. As the next step, the system of equations for the program will be assembled to compute the possible abstract values of x in each program point. But first we have to define the following two operators:

- $+ : \langle x, x' \rangle + \langle y, y' \rangle = \langle x + y, x' + y' \rangle$
- $\cap : \langle x, x' \rangle \cap \langle y, y' \rangle = \langle max(x, y), min(x, y) \rangle$

The equations are following:

1. $C_0 = \langle , \rangle$
2. $C_1 = \langle 1, 1 \rangle$
3. $C_2 = C_1 \circ_I C_4$

4. $C_3 = C_2 \cap \langle -\infty, 100\rangle$

5. $C_4 = C_3 + \langle 1, 1\rangle$

6. $C_5 = C_2 \cap \langle 101, \infty\rangle$

The semantics of $C_1 = \langle 1, 1\rangle$ *is that the variable* x *has a value from interval* $\langle 1, 1\rangle$ *at the program point* C_1*. The semantics of the other equations are analogical.*

If we start computation of fixpoint using just $\circ_I$ *operator the computation takes* 101 *steps before the fixpoint is reached. That is maximal number of iterations of the* `while` *cycle. The positive message is that the computation will terminate, the negative one is that it will take quite a long time for such a simple program. Therefore we introduce widening* ∇ *defined as follows:*

- $\langle , \rangle$ *is the null element of* ∇
- $\langle x, x'\rangle \nabla \langle y, y'\rangle = \langle (x < y)\,?-\infty : x, (y' > x')\,?\,\infty : x'\rangle$

where ? : *is a ternary operator with the same semantics as in the C language.*

The equitation 3 *will be modified using the widening:* $C_2 = C_2 \nabla (C_1 \circ_I C_4)$*.*

The approximation of solution is computed in the following iterations:

0. iteration:

- $\forall i \in \{0, \ldots, 5\} : C_i = \langle , \rangle$

1. iteration:

- $C_1 = \langle 1, 1\rangle$
- $C_2 = \langle 1, 1\rangle$
- $C_3 = \langle 1, 1\rangle$
- $C_4 = \langle 2, 2\rangle$

2. iteration:

- $C_2 = \langle 1, 1\rangle \nabla (\langle 1, 1\rangle \circ_I \langle 2, 2\rangle) = \langle 1, 1\rangle \nabla \langle 1, 2\rangle = \langle 1, \infty\rangle$
- $C_3 = \langle 1, 100\rangle$
- $C_4 = \langle 2, 101\rangle$
- $C_5 = \langle 101, \infty\rangle$

Another iteration would not change anything so the fixpoint is reached. The widening operator definitely accelerated the computation and saved 98 *iterations. On the other hand, the information about values at points in* C_2 *and* C_5 *is very rough due to abstraction.*

The loss of information can be reduced by narrowing operator $\triangle$ *with this definition:*

- $\langle , \rangle$ *is the null element of* $\triangle$
- $\langle x, x'\rangle \triangle \langle y, y'\rangle = \langle (x == -\infty)\,?\,y : min(x, y), (x' == \infty)\,?\,y' : max(x', y')\rangle$

We update 2. *equitation using narrowing:* $C_2 = C_2 \triangle (C_1 \circ_I C_4)$*. The computed solution of equations will be refined in the following iterations:*

3. iteration:

- $C_2 = C_2 \triangle (C_1 \circ_I C_4) = \langle 1, \infty\rangle \triangle (\langle 1, 1\rangle \circ_I \langle 2, 101\rangle) = \langle 1, \infty\rangle \triangle \langle 1, 101\rangle = \langle 1, 101\rangle$
- $C_3 = \langle 1, 101\rangle \cap \langle -\infty, 100\rangle = \langle 1, 100\rangle$
- $C_5 = \langle 1, 101\rangle \cap \langle 101, \infty\rangle = \langle 101, 101\rangle$

After the narrowing the information is much more precise and only one iteration was needed so 97 *iterations were saved compared to the computation without widening and narrowing.*

Another advantage of analysis of a program in abstract domain is a compact representation using intervals instead of set of integer values in concrete domain. E.g., the used abstraction (widening operator) would lead to manipulation with an infinite set.

As we mentioned, the framework of abstract interpretation is much more general and the preceding example is just a special case. Many other static analysis can be formulated as an abstract interpretation, e.g., data flow analysis, pointer analysis, or correct array access analysis.

3 Forest Automata

This section gives an introduction to forest automata. First, the definition of the tress and graphs are given. Then the automata accepting them are defined. The goal is to introduce the automata in a general way but with respect to the aspects needed for abstract interpretation. The structure and the definitions in this section are based on [9].

3.1 Graphs, Trees and Forests

In the following text we use $dom(f)$ to denote a domain of a total mapping $f : A \to B$. The range of such mapping is denoted by $range(f)$. A symbol a_i is the i-th symbol of a word $w = a_1 \cdots a_n$ over an alphabet Σ.

We need for purposes of graphs introduce a *ranked alphabet*. Intuitively, it is an alphabet with symbols having an integer rank. When a symbol with a given rank n appears in a labelled graph then it is required for a symbol to be over exactly n edges leading from one node to n successors. Formally, a *ranked alphabet* is a pair $(\Sigma, \#)$ where Σ is a finite set of symbols and $\# : \Sigma \to \mathbb{N}$ is a related mapping assigning to a symbol its rank. We use for brevity just Σ to denote a ranked alphabet. A (directed, ordered, labelled) *graph* is a total map $g : V \to \Sigma \times V^*$ where V is a finite set of nodes. We call the symbols from Σ *labels* in context of the graphs. The map g maps each node $v \in V$ to:

1. a label $a \in \Sigma$ that we denote by $l_g(v)$,
2. a sequence of *successors* $(v_1 \cdots v_n) \in V^n$ for $n \in \mathbb{N}$. We denote successors by $S_g(v)$ and v_i is denoted by $S^i_g(v)$. Symbol $l_g(v)$ is such that $\#(l_g(v)) = |S_g(v)|$.

A pair $v \mapsto (a, v_1 \cdots v_n)$ where $v, v_1, \ldots, v_n \in V$, $a \in \Sigma$ such that $g(v) = (a, v_1 \cdots v_n)$, is called an *edge* of g. A *leaf* of g is a node $v \in V$ without successor so it holds $S_g(v) = \epsilon$. Consider set of all sequences of successors such that a node $v' \in V$ appears among the successors, then the cardinality of this set is the *in-degree* of a node $v' \in V$ (denoted by $idg_g(v')$). More intuitively, the in-degree of a node is a number of the incoming edges to the node. Formally, in-degree is defined as $idg(v') = |\{(v \mapsto (a, v_1 \cdots v_n), i)) \mid v \mapsto (a, v_1 \cdots v_n)$ is an edge such that $i \in \{1, \ldots, n\} : v' = v_i\}|$. When a node $v' \in V$ has $idg(v') > 1$ it is called a *join*.

A special case of a graph is a *tree*. Informally, a tree is a graph where each node is a successor of at most one of the other nodes. The node which is not a successor of any node is called *root*.

Formally, t is a graph which is either empty, or it has exactly one root and $\forall v \in V : idg(v) \leqslant 1$ (each node is a successor of at most one of the other nodes).

A *path* from $v \in V$ to $v' \in V$ is a sequence $p = v_0, i_1, v_1, \ldots, i_n, v_n$ where $v = v_0, v' = v_n$ and $\forall j \in \{1, \ldots, n\} : v_j = S^{i_j}(v_{j-1})$ (v_j is the i_j-th successor of v_{j-1}). The *length* of the path p is n what is denoted by $length(p) = n$. The path $p = v_0, i_1, v_1, \ldots, i_n, v_n$ is acyclic if $\forall v_i, v_j \in p : i \neq j \Rightarrow v_i \neq v_j$. The *cost* of the acyclic path p is the sequence $i_1, \ldots, i_n$. The path p is *cheaper* than path p' iff the cost of p is lexicographically smaller than that of p'. A node $u \in V$ is *reachable* from a node $v \in V$ iff there exists a path from v to u or $u = v$. A node $u \in V$ is a *root* of the graph g iff all nodes $v \in V$ are reachable from u and u is not successor of any other state. We alternatively determine a root of a graph by a mapping $root : g \rightarrow V$ which maps a graph to its root.

An example of a graph with $V = \{v_1, v_2, v_3, v_4, v_5\}$ and the alphabet $\Sigma = \{a, b\}$ with a ranking function $\#$ such that $\#(a) = 2$ and $\#(b) = 0$ is in Figure 1. This graph is a tree.

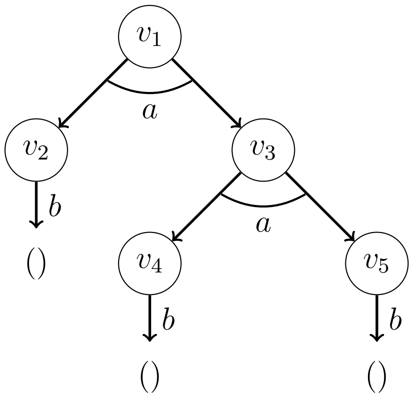

Figure 1: A graph t that has attributes of a tree.

In this section we assume without the loss of generality that $\Sigma \cap \mathbb{N} = \emptyset$. A Σ-labelled *forest* is a sequence of trees $t_1 \cdots t_n$ over $(\Sigma \cup \{1, \ldots, n\})$ where $\forall i \in \{1, \ldots, n\} : \#i = 0$ (the rank of new symbols from $\mathbb{N}$ is zero). We suppose that the sets of nodes of the trees $t_1, \ldots, t_n$ are disjoint. *Root references* are leaves labelled by a label $i \in \{1, \ldots, n\}$. The newly added symbols from $\{1, \ldots, n\}$ labelling root references are supposed to interconnect trees in a forest. E.g., when the second tree has a leaf v which is a root reference labelled by 1 then it is a symbolical connection with the root of the first tree. The forest $t_1 \cdots t_n$ represents a graph $\otimes t_1 \cdots t_n$ that arises by interconnecting roots by the related root reference. Formally, the graph $\otimes t_1 \cdots t_n$ contains an edge $v \mapsto (a, v_1 \cdots v_m)$ iff $\exists t_i \in \{t_1, \ldots, t_n\} : \exists (v \mapsto (a, v'_1 \cdots v'_m)) \in edges(t_i) : \forall j \in \{1, \ldots, m\} : v_j = h(v'_j)$ where $edges(t_i)$ is the set of all edges of the tree t_i and

$$h(v'_j) = \begin{cases} root(t_k) & \text{if } v'_j \text{ is a root reference with } l(v'_j) = k \\ v'_j & \text{otherwise.} \end{cases}$$

Consider a forest f over $\Sigma \cup \{\overline{2}, \overline{3}\}$ (Σ is the same one as in Figure 3.1) in Figure 2. The graph $\otimes t_1, t_2, t_3$ obtained by interconnecting trees in the forest f is in Figure 3.

We mark some graph node as *input* or *output* one what is later needed for manipulation with forest automata of higher degree. The marking of input and output port is done by tuple of so-called *ports* in the following way. An *input-output-graph* (io-graph) is a pair (g, ϕ) (also denoted by g_ϕ) where g is a graph and $\phi = (\phi_1 \cdots \phi_n) \in dom(g)^+$ is a sequence of ports, where ϕ_1 is the input port and $\phi_2 \cdots \phi_n$ are output ports. The ports are unique in ϕ. The graph g_ϕ is called *accessible* if its root is the input port ϕ_1.

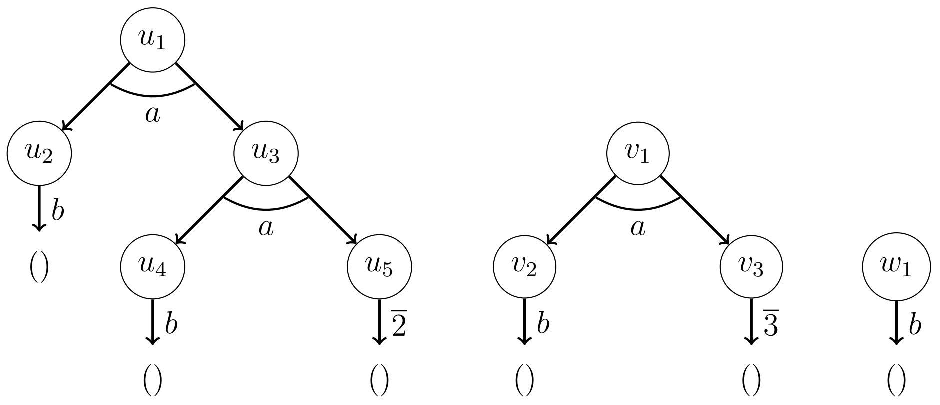

Figure 2: A forest f consisting 3 trees t_1, t_2, t_3 with roots u_1, v_1, w_1.

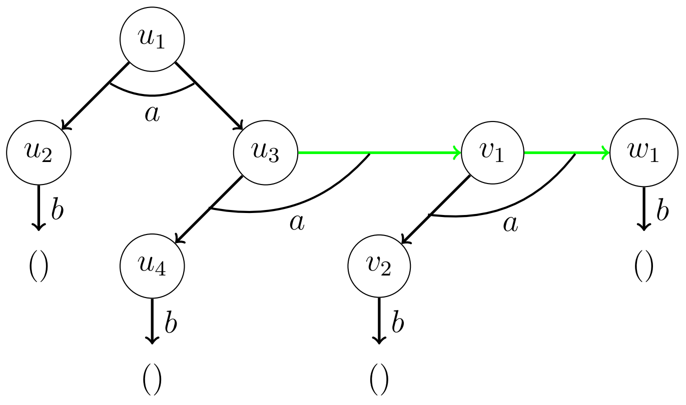

Figure 3: The graph $\otimes t_1, t_2, t_3$ obtained from the forest f in Figure 2. The green edges are the ones that were added to create the graph from the forest f.

The *cut-points* $cps(g_\phi)$ of a graph g_ϕ are ports and joins of the graph, i.e. $cps(g_\phi) = \{v \in V \mid v \in \phi \vee idg(v) > 1\}$.

An *io-forest* is straightforward extension of a forest with concept of input and output ports. We add a sequence of numbers which denote the trees of the forest which roots would be input and output of an interconnected graph of the io-forest. Formally, an *io-forest* is a pair $f = (t_1 \cdots t_n, \pi)$ such that $n \geqslant 1$ and π is a sequence of port indices consisting of numbers $\pi_i \in \{1, \ldots, n\}$ where $1 \leqslant i \leqslant n$. The port index π_1 is an index of input port and $\pi_2 \ldots \pi_{|\pi|}$ is a sequence of the indices of the output ports. The indices are again unique.

The io-graph $\otimes f$ is constructed from a forest f by making the roots of the trees indexed by numbers from π input and output graphs of $\otimes f$, i.e. $\otimes f = (\otimes t_1 \cdots t_n, root(t_{\pi_1}), \ldots, root(t_{\pi_n}))$.

Because the concept of the io-graphs and io-forests may be a little bit hard to understand we provide with an example for better explanation.

Example 3.1 *In this example, we refer to the graph t from Figure 1. It can be extended to an io-graph t_ϕ by adding ports $\phi = (v_1, v_4, v_5)$. The resulting io-graph t_ϕ has the input port v_1 and the output ports v_2, v_3. Because v_1 is the root, the graph t_ϕ is accessible. The cut-points of t_ϕ are v_1, v_4, v_5.*

Now consider the forest f in Figure 2. This forest could be extended to an io-forest $f_{io} = ((t_1, t_2, t_3), \pi)$ by defining a sequence of the port indices π which could be e.g., $(1, 3)$. Then the *graph $\otimes f_{io}$ is a pair $(\otimes(t_1, t_2, t_3),\ (u_1, w_1))$. The graph $\otimes(t_1, t_2, t_3)$ is the same as in Figure 3. The input port u_1 is indeed $root(t_1)$ and the output port w_1 because it is the root of the third tree in f and the index of the output port is* 3.

It is needed for efficient manipulation with forests to define minimality and canonicity which clear the way for deterministic representation of forests. An io-forest $f = (t_1 \cdots t_n, \phi)$ with the underlying graph $\otimes f$ is *minimal* iff the roots of the trees $t_1, \ldots, t_n$ correspond to the cut-points of $\otimes f$. In the case of minimal io-forest, there exists a bijection between $\{root(t_k) \mid t_k \in \{t_1, \ldots, t_n\})\}$ and $cps(\otimes f)$. Therefore io-forest f is an unique representation of $\otimes f$ up-to to permutations of $t_1, \ldots, t_n$.

Also the permutations of $t_1, \ldots, t_n$ can lead to grown of complexity and so we need to reduce also this source of non-unique representation of an io-graph by io-forest. Therefore we define a cannonical ordering over cut-points of $\otimes f$ in the following way. A relation $\preceq_p \subseteq cps(\otimes f) \times cps(\otimes f)$ is *the canonical ordering* iff $c_1 \preceq_p c_2 \Leftrightarrow$ the cost of the cheapest path from ϕ_1 (input port) to c_1 is *smaller* than the cost of the smallest path from ϕ_1 to c_2. Now, it is finally possible to define also cannonical representation of io-graph by an io-forest. The io-forest f_c is *canonical* iff it is minimal, the trees $t_1, \ldots, t_n$ are ordered by $\preceq_p$, and $\otimes f$ is accessible. The canonical io-forest is a unique representation of an accessible io-graph and can be obtained by a depth-first traversal (DFT) of $\otimes f$.

We need to assume that there is an ordering $\leqslant_\Sigma$ over labels Σ of $\otimes f$ to make DFT deterministic. The DFT starts with a stack consisting of the input and output ports ordered by $\preceq_p$. The DFT is run over $\otimes f$. The tuples of successors of a node are visited in the order given by $\leqslant_\Sigma$ The nodes in these tuples are also ordered. The result of DFT is a canonical forest f_c with the following cannonical ordering of the trees. The first ones are trees corresponding to the ports of f ordered by $\preceq_p$ and the rest of the trees is ordered by order in the DFT traversal visit.

3.2 Automata over Trees and Forests

We have defined trees which can been seen as an analogy to words. Words are accepted by finite automata and the computational model working over trees is a tree automaton which we define now.

A (finite, non-deterministic, top-down) *tree automaton* (TA) is a quadruple $A = (Q, \Sigma, \Delta, R)$ where

- Q is a finite set of *states*,
- Σ is a ranked alphabet,
- Δ is a set of *transition rules* where transitions have a form $(q, a, q_1 \cdots q_n)$ where $q, q_1, \ldots, q_n \in Q$, $a \in \Sigma$, $n \geqslant 0$ and $\#a = n$. Alternatively, we write $q \xrightarrow{a} (q_1 \cdots q_n)$ to denote that $(q, a, q_1 \cdots q_n) \in \Delta$. The rule is called *leaf rule* when $n = 0$,
- $R \subseteq Q$ is a set of *root states.*

TA has the following semantics. A *run* of A over a tree t is mapping $\rho : dom(t) \to Q$ such that $\forall v \ \in dom(t)\ \exists q \xrightarrow{a} (q_1 \cdots q_n) \in \Delta : q = \rho(v) \wedge \forall i \in \{1, \ldots, |S(v)|\} : q_i = \rho(S(v)_i)$. We use $t \Rightarrow_\rho q$ to denote that ρ is a run of A over a tree t s.t. $\rho(root(t)) = q$ and $t \Rightarrow q$ denotes that there exists ρ s.t. $t \Rightarrow_\rho q$. A state $q \in Q$ has language defined as $L(q) = \{t \mid t \Rightarrow q\}$ and the *language* of A is defined as $L(A) = \bigcup_{q \in R} L(q)$.

Example 3.2 *Consider a TA $A = (Q, \Sigma, \Delta, R)$ where $Q = \{q_1, q_2, q_3, q_4, q_5\}$, $\Sigma = \{a, b\}$, such that $\#(a) = 2, \#(b) = 0$, $R = \{q_1\}$, and $\Delta = \{q_1 \xrightarrow{a} (q_2, q_3), q_2 \xrightarrow{b} (), q_3 \xrightarrow{b} (q_4, q_5), q_4 \xrightarrow{b} (), q_4 \xrightarrow{b} ()\}$.*

Then the map ρ such that $\forall i \in \{1,\dots,5\} : \rho(v_i) = q_i$ is a run A over the tree t from Figure 1. Since $\rho(root(t)) = \rho(v_1) = q_1$ then $t \in L(q_1)$ and because $q \in R$ it also holds $t \in L(A)$.

In the same way we have extended trees to forest we define forest automaton accepting forests consisting of trees.

A *forest automaton* (FA) over alphabet Σ is a pair $F = (A_1 \cdots A_n, \pi)$ where $A_1 \cdots A_n$ is a sequence of tree automata defined over the alphabet $\Sigma \cup \{1,\dots,n\}$ and $\pi = I_1 \cdots I_n$, where $I_1,\dots,I_n \in \{1,\dots,n\}$ is a sequence of port indices. Forest automata have two different languages. The first one is related to the notion of io-forests and the other one to io-graphs. The first one is the so-called *forest language* obtained by Cartesian product of the languages of particular TA (and port indices) of FA so the forest language is a set of the io-forests. The other is the so-called *graph language* obtained by connecting the io-forests from the forest language to io-graphs. We define the *forest language* of the FA F as the set of io-forests $L_f(F) = L(A_1) \times \ldots \times L(A_n) \times \{\pi\}$. As we mentioned the forest language contains also port indices otherwise the items of language would not be io-forests. The graph language would not be also reconstructable without ports in the forest language. The *graph language* of F is the set of io-graphs $L(F) = \{\otimes f \mid f \in L_f(F)\}$. FA F respects *canonicity* if $\forall f \in L_f(F) : f$ *is canonical.*

The section about abstract interpretation introduced importance of fixpoint computation in this technique. The crucial operation for checking whether an abstract interpretation reached fixpoint of possible abstract values in a given program point is checking ordering between two abstract values. The ordering is checked in case of forest automata by checking inclusion of their languages (because their languages represent possible shapes of the heap in the given program point). The inclusion between languages of two forest automata that respect canonicity is checked *component-wise*, i.e. checking language inclusion of their tree automata one by one, what comes from the following theorem.

Let $F^1 = (A^1_1 \cdots A^1_{n_1}, \pi^1)$ and $F^2 = (A^2_1 \cdots A^2_{n_2}, \pi^2)$ be two canonicity respecting FA. Then $L(F^1) \subseteq L(F^2)$ iff

- $n_1 = n_2$
- $\pi^1 = \pi^2$
- $\forall i \in \{1,\dots,n_1\} : L(A^1_i) \subseteq L(A^2_i)$

Proof can be found in [6].

Example 3.3 *Consider the io-forest $f_{io} = ((t_1,t_2,t_3),(1,3))$ from Example 3.1. The tree t_1 (which is the same as the tree t) belongs to the language of TA A defined in Example 3.2. We further have a TA $B = (Q_B, \Sigma, \Delta_B, R_B)$ where $Q_B = \{p_1,p_2,p_3\}$, Σ is same as in Example 3.2, $\Delta = \{p_1 \xrightarrow{a} (p_2,p_3), p_2 \xrightarrow{b} (), p_3 \xrightarrow{b} ()\}$ and $R = \{p_1\}$. The TA B contains t_2 in its language. Finally, consider a TA $C = (Q_C, \Sigma, \Delta_C, R_C)$ where $Q_C = \{r_1\}$, Σ is again the same as before, $\Delta = \{r_1 \xrightarrow{b} ()\}$ and $R = \{r_1\}$. A set $L(C)$ contains t_3. Putting all automata together we can construct the forest automaton $F = ((A,B,C),(1,3))$. The io-forest f_{io} is in the forest language $L_f(F)$ of F because it belongs to $L(A) \times L(B) \times L(C) \times \{(1,3)\}$. Hence the graph $\otimes f_{io} = (\otimes(t_1,t_2,t_3),(u_1,w_1))$ (shown in Example 3.1) is in the graph language $L(F)$.*

3.3 Forest Automata of Higher Level

The already defined forest automata are able to model basic data structures such as singly-linked lists or trees without parent pointers. In more general terms, they are able to represent data structures with bounded number of cut-points. If we want to represent also dynamic data structures with unbounded number of cut-points such as doubly-linked lists or skip-lists it is necessary to extend expressive power of the basic FA. This is done introducing hierarchical FA what are FA having FA of lower level as the symbols on their edges. These hierarchical FA are defined in this section but to make their definition easier we firstly define structured labels.

Let Γ be a ranked alphabet of sub-labels with defined total ordering $\sqsubset$ which is called *sub-labels ordering*. Let g be a graph defined over 2^Γ where $A \in 2^\Gamma$ denotes a label of g such that $\forall A \subseteq \Gamma : \#A = \sum_{a \in A} \#a$. These symbols $A \in 2^\Gamma$ are *structured labels*. The graph g has edges in the form $v \mapsto (A, v_1 \cdots v_n)$ where $A \subseteq \Gamma$ and $\#A = n$. We denote such edge by e. An edge e consists of *sub-edges* forming sequence $e\langle 1 \rangle = v \ \mapsto (a_1, v_1 \cdots v_{\#a_1}) \cdots e\langle n \rangle = v \ \mapsto (a_m, v_{n-\#a_m+1} \cdots v_m)$. The i-th sub-edge of e in g is denoted by $e\langle i \rangle = v \ \mapsto (a_i, v_k \cdots v_l)$ where $i : 1 \leqslant i \leqslant m$. We use $SE(g)$ to denote all sub-edges of the graph g. A node v of a graph is *isolated* if it is not part of any sub-edge. Formally, a node v is isolated iff $\nexists e\langle i \rangle = v \ \mapsto (a_i, v_k \cdots v_l) \wedge \nexists e\langle i \rangle = v' \mapsto (a_i, v_k \cdots v_l) : v \ \in \{v_k, \ldots, v_l\}$. A graph g is *unambiguously determined* by $SE(g)$ if g has no isolated nodes.

Now tree and forest automata will be extended to work over the defined structured labels.

A TA over structured labels is quadruple $A = (Q, 2^\Gamma, \Delta, R)$ where Q, Γ and R has the same meaning as in the case of basic TA and Δ is a set of transition rules with the rules in the form $(q, \{a_1, \ldots, a_m\}, q_1 \cdots q_n)$ where $q, q_1, \ldots, q_n \in Q$, $\{a_1, \ldots, a_m\} \in \Gamma$. Each rule could be interpreted as a sequence of the *rule-terms* $d\langle 1 \rangle = q \mapsto (a_1, q_1 \cdots q_{\#a_1}) \cdots d\langle n \rangle = q \mapsto (a_m, q_{n-\#a_m+1} \cdots q_n)$ and we denote the i-th rule term of sequence again by $d\langle i \rangle$ where $i \in \{1, \ldots, m\}$,

The hierarchy FA over structured labels will be defined in the inductive way starting from the forest automata of *level* 1. Forest automata of level 1 have the structured labels from 2^Γ but have not other forest automata in their labels. These FA form the set Γ_1. Forest automata of *level i* form the set Γ_i. A forest automaton F of level $i+1$ is defined over the ranked alphabet $2^{\Gamma \cup \Delta}$ where Δ is a subset of forest automata of level i which are called *boxes* of F. The set of all forest automata of all levels $\sum_{i \geqslant 0} \Gamma_i$ is denoted by Γ^* and it is ordered by a total ordering $\sqsubset_{\Gamma_*}$ defined using ordering $\sqsubset$ for sub-labels.

The semantics of forest automata of higher level are defined using an operation *sub-edge replacement*. This operation does a replacement of an edge with graph in label by this graph. The operation basically removes a sub-edge and connects to the parent and successors of the removed sub-edge a graph labelling the sub-edge using input and output ports of the graph for this connection.

Formally, let g be a graph with an edge $e \in edges(g)$ and sub-edge $e\langle i \rangle = v_1 \rightarrow (a, v_2 \cdots v_n)$. Let g'_ϕ be an io-graph such that $|\phi| = n$ and assume that $dom(g) \cap dom(g') = \emptyset$. The sub-edge $e\langle i \rangle$ could be replaced by g' if $\forall j \in \{1, \ldots, n\} : l_g(v_j) \cap l_{g'}(\phi_j) = \emptyset$ (this conditions checks whether there is no successor of v_j and ϕ_j reachable over the same label from the both nodes). The result of this operation is a graph $g[g'_\phi / e\langle i \rangle]$ which can be computed using the following algorithm:

- $SE(g_0) = SE(g) \cup SE(g') \backslash \{e\langle i \rangle\}$.
- $\forall j \in \{1, \ldots, n\}$: *the graph g_j is obtained from g_{j-1} by following procedure*
    1. Deriving a graph h by replacing the origin of the sub-edges of the j-th port ϕ_j of g' by v_j.
    2. Redirecting edges from ϕ_j to v_j, i.e., replacing all occurrences of ϕ_j in $range(h)$ by v_j
    3. Removing ϕ_j.
- Graph g_n is the result (so the graph $g[g'_\phi / e\langle i \rangle]$ is the same one as g_n).

We used the introduced operation of sub-edge replacement to define the similar methods over forest automata.

- *Unfolding* of a graph g is replacement of one of its sub-edges with a symbol, which is an FA F', by a graph from language $L(F')$ of this FA. Formally, consider an sub-edge $e\langle i\rangle$ of the graph g with a symbol a, a is a box containing an FA F' and $g'_\phi \in L(a)$. Then $h = g[g'_\phi/e\langle i\rangle]$ is an unfolding of g. We use $g \prec h$ to denote that h is unfolding of g.

- *Folding* is a replacement of g'_ϕ by $e\langle i\rangle$ in h obtaining g. So we can say that g'_ϕ is folded to $e\langle i\rangle$.

A transitive reflective closure of $\prec$ is denoted by $\prec^*$. A set of all graphs obtained by repeated application of unfolding from a graph g over ranked alphabet Γ is called Γ*-semantics* and is denoted by $[\![g]\!]_\Gamma$ or just simply $[\![g]\!]$ when the alphabet is obvious from context. Formally defined, Γ-semantics is the set of graphs g' such that $g \prec^* g'$. Finally, Γ*-semantics* is defined for a FA F of higher level as $[\![F]\!] = \bigcup_{g_\phi \in L(F)}([\![g]\!] \times \{\phi\})$.

The extension of canonicity to automata of higher level is straightforward. A FA F is canonicity respecting if $\forall f \in L_f(F) : f$ *is canonical.* The language inclusion checking is again possible in the component-wise way like in the case of basic FA, as it is proved in [7]. However, the testing language inclusion with the same algorithm used for the non-hierarchical FA is sound but incomplete because semantics of the structured labels is omitted and they are treated like non-structured symbols.

4 Forest Automata as an Abstract Domain

This section connects abstract interpretation and forest automata to a working verification method for shape analysis. The method needs to represent dynamic data structures using forest automata as an abstract domain and also to model commands manipulating dynamic data structures as abstract transformers over forest automata. Both issues will be covered in this section together with other parts of abstract interpretation such as join or widening over forest automata.

4.1 Heaps and Forests

So far we use notion of heap very informally but it is necessary to define it in more technical terms for description of forest automata as an abstract domain. Consider a heap with allocated dynamic data structures. The heap can be viewed as a graph where the nodes of graph represent allocated memory cells. The edges between nodes of graph does not represent the pointers in data structures as one might expect but these pointers are represented by the structured labels over the edges. The elements of structured labels are of the two kinds. The first ones are so called *pointer selectors* forming set $PSel$ representing the mentioned pointers connecting two memory nodes. The other ones are so called *data selectors* forming set $DSel$ representing values of data domain $\mathbb{D}$ stored in a memory node. The labels are from the set 2^Γ where $\Gamma = PSel \cup DSel \times \mathbb{D}$. Pointer selectors also contain two special values — *null* and *undefined*. Data selectors are captured in the labels with data pointed by them what is reflected by $DSel \times \mathbb{D}$ in definition of Γ. A structured label over the edge leading from a node of a heap graph models a content of an allocated memory represented by the node.

The graphs with unbounded number of cut-points are represented by hierarchical forest automata by folding a repeating part of graph causing unboudness into a box. Further we extend the graphs to io-graphs where the ports of io-graph are nodes pointed by a variable (i.e. a memory node pointed by a pointer variable).

In order to represent an io-graph modelling heap by forest automata, we need to decompose such io-graph to a forest what is done in the following way. First, the cut-points of the io-graph are identified and numbered by a DFS traversal of the graph starting from the nodes pointed by the variables. Then the graph is split in these cut-points. The splitting creates a set of trees with the cut-points as their roots. We label the edges leading to the cut-points before splitting by root references with a number assigned to the cut-point by the DFS traversal. Finally, we have a tuple of trees interconnected by root references. The trees in tuples should be ordered by numbers assigned to cut-points in the DFS traversal. Since the tuples of trees can be accepted by forest automata it is possible to represent set of heap io-graphs by forest automaton.

4.2 Forming Abstract Interpretation

When we described correspondence between heaps and forest automata it is possible to formally define concrete and abstract domain. The concrete domain is for each program point a set of pairs (σ, H) where σ is a mapping assigning each variable to a node in heap graph, to *null* or to *undefined*, and H is a heap io-graph representing data structures allocated on graph. The abstract domain is for each program point a set of pairs (σ, F) where σ is a mapping assigning each variable to a TA in F, to *null* or to *undefined*, and F is a forest automaton representing set of heap io-graphs.

All possible abstract configurations for each program point are computed by iterative application of abstract transformer until reaching fixpoint. The unions of sets obtained by application of abstract transformers are used as a join at the junctions. At the loop points, the widening is also applied. The widening is done by abstractions introduced in *abstract regular model checking* [2]. The abstraction is used for each TA of FA separately. So far, two kinds of abstraction were used.

The first is so called *height abstraction.* This abstraction merges states of a TA T accepting the trees with the same prefixes up to height k where k is chosen constant. Formally, $q \equiv q' \Leftrightarrow L(q)_{\leqslant k} = L(q')_{\leqslant k}$ where $L(p)_{\leqslant k} = \{t' \mid t \in L(q) \wedge t' \textit{ is obtained from } t \textit{ by restriction up to height } k\}$ and states p, q, q' are states of TA T.

Another kind of abstraction is so called *predicate abstraction.* A set of predicates $\mathcal{P} = \{P_1, \ldots, P_n\}$ is needed for this abstraction. The states in TA merged by this abstraction are the one which have an non-empty intersection with the same set of predicates. Formally, let $T = (Q_T, \Sigma_T, \Delta_T, R_T)$ be a TA then $q \equiv q' \Leftrightarrow (\forall P \in \mathcal{P} : L(q) \cap P \neq \emptyset \Leftrightarrow L(q') \cap P \neq \emptyset)$ where q, q' are states of TA T. The algorithm for computing states by merging is following. The product of TA T with each of TA representing predicates $\mathbb{P}$ is created. The states of TA T are in the product states of the created product tree automata with the different states of TA representing predicates. The states of TA T which are in pairs of the product states with the same states are merged by the abstraction. Let formalize this idea. Consider the set of predicates $\mathcal{P}$ represented by the set of TA $T = \{A_1^{\mathcal{P}}, \ldots, A_m^{\mathcal{P}}\}$. Each of product automata $T \times A_i^{\mathcal{P}}$, where $1 \leqslant i \leqslant n$, has the state set consisting of the product states of the form $(p, q) \in Q_T \times Q_{A_i^{\mathcal{P}}}$. Consider the auxiliary function m labelling each state of A by a set of states of the predicate tree automata, i.e., the function $m : Q_T \rightarrow 2^{Q_{A_1^{\mathcal{P}}} \cup \ldots \cup Q_{A_m^{\mathcal{P}}}}$ is defined as follows: $m(p) = \{q \in Q_{A_1^{\mathcal{P}}} \cup \ldots \cup Q_{A_m^{\mathcal{P}}} \mid \exists\, T \times A_i^{\mathcal{P}} : (p, q) \in Q_T \times Q_{A_i^{\mathcal{P}}}\}$. The equivalence relation $\equiv \subseteq Q \times Q$ is defined: $(p_1, p_2) \in \equiv \Leftrightarrow m(p_1) = m(p_2)$.

When the widening is done at a loop point the language inclusion between FA before and after widening is performed to check whether a fixpoint has been reached. The widening is repeated until the fixpoint is reached. Folding over FA and normalization is also done after each widening (and before checking inclusion) to bounded number of cut-points and to obtain unique representation of FA before inclusion checking. If folding would not be done here then an unbounded number of cut-points may arise.

The weakness of forest automata approach to abstract interpretation is absence of the narrowing operation. This comes from the fact that forest automata based shape analysis has properties of abstract interpretation but it was mainly designed on the basis of abstract regular model checking. Therefore when a spurious error is found in program (error that is not presented in program but was caused by overapproximating abstraction) then it is necessary to restart the whole abstract interpretation with a refined abstraction. The height abstraction is refined by increasing the height k and the predicate abstraction is refined by a predicate learned from a spurious counterexample [9]. The predicate abstraction approach is more precise because a learned predicate prevents reaching exactly the spurious error and it does not weaken abstraction too much.

4.3 Abstract transformers

The chosen abstract transformers will be described in this section. The description is mainly based on [1]. We will cover abstract transformers modelling concrete transformers manipulating heap structures. The description given here will be more abstract compare to real implementation in the *Forester* tool [10] which implements shape analysis based on forest automata. In Forester, one concrete transformer is translated to more abstract transformers which are implemented as instructions of Forester microcode. However, the description of the microcode would be full of technical details darkening the essence of abstract transformers, therefore the abstract transformers will be presented from the perspective of their effects on abstract domain omitting implementation details.

- `malloc(size)` — Allocates a new memory node with given *size*. In abstract domain, this is done by creating a new tree automaton which is added to forest automaton representing current heap. The new tree automaton has one state with transition labelled by structured label containing pointer selector x together with information about size of allocated memory.

- `free(x)` — Takes a node of TA pointed by x and isolates all its selectors in labels of transition under this node. The isolation consists in creating a new TA for representation of a memory pointed by a selector being isolated and replacing the original selector by root reference to the new TA. Finally, the node pointed by the variable x is removed. Notice that if one of the newly created tree automata is not accessible from any other node then a garbage is created.

- `x = y->sel` — When the selector `y->sel` leads to a node n which is root reference then the variable x will be redirected to root of the referenced TA. I.e., $\sigma(x)$ is changed to the referenced TA in abstract domain. Otherwise, a node n in TA t pointed by selector `y->sel` is isolated to a new TA containing part of TA t reachable from the node n. The original occurrence of the node n is replaced by root reference to the new TA which is added to FA representing heap. Then the variable x will point to this new TA. It is possible that the selector `y->sel` is folded in a box before the operation. Then it is necessary to perform unfolding to access the selector.

- `x->sel = y` — Firstly, the node pointed by the selector `x->sel` is isolated to a new TA as in the previous case. The node pointed by y is also isolated. Then the node pointed by `x->sel` is replaced by root reference to TA pointed by y (note, that y is definitely pointing to a TA which was created during the isolation or was pointed by y without need of isolation). The unfolding may also be needed in this transformer for the same reasons as in the previous one.

- `x = y` — The value $\sigma(x)$ is updated in abstract domain to contain the same value as $\sigma(y)$.

- `x = *y` — Assume that x is the base type of pointer y. Then a value get by evaluation of $\sigma(y)$ in abstract domain is stored to x.

It is also necessary to perform garbage checking after each of the described operations. That is done by checking whether all tree automata in FA representing heap are accessible from nodes pointed by pointer variables.

5 Conclusion

This work presented join of abstract interpretation and forest automata for the purposes of shape analysis. First, the theory of abstract interpretation was presented then forest automata were covered. Finally, a combination of both and its application in shape analysis were described.

Forest automata have capability to model complex data structures and it is possible to analyse program manipulating these data structures such as skip-list of the second and the third level, trees with parent pointers or doubly-linked list [8].

The biggest advantages of this approach is its flexibility and scalability. It is able automatically verify different data structures without need to provide any specific information manually. Forest automata based verification beats separation logic methods for shape analysis in this aspect because they need often manually crafted predicates solving corner cases in verification of different data structures. Scalability of forest automata lies in an ability to make a local change in a heap without need to propagate this change all over an automaton what can lead to high computational complexity. This is in contrast to abstract regular model checking which misses this scalability feature.

On the other hand, the disadvantage of forest automata based verification is current lack of the more precise abstraction refinement. The mentioned predicate abstraction is currently designed only for non-hierarchical forest automata what disable a verification of data structures such as red-black trees or b+ trees. The lack of narrowing also leads to necessity of restarting the analysis when a spurious counterexample is found.

The possible development of the forest automata based shape analysis is in resolving the mentioned weaknesses but also in improving the scalability by introducing methods for modular verification of parts of large systems without explicit knowledge about the rest of system. It is also needed to improve an implementation of the method because its current state in the Forester tool is far from maturity.

References

[1] Parosh Aziz Abdulla, Lukáš Holík, Bengt Jonsson, Ondřej Lengál, Cong Quy Trinh, and Tomáš Vojnar. Verification of Heap Manipulating Programs with Ordered Data by Extended Forest Automata. In *Proc. of ATVA 2013*, volume 8172, pages 224–239. Springer International Publishing, 2013.

[2] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. Abstract Regular (Tree) Model Checking. *International Journal on Software Tools for Technology Transfer*, 14(2):167–191, 2012.

[3] Patrick Cousot. Semantic Foundations of Program Analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[4] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model For Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL 1977*, pages 238–252. ACM, 1977.

[5] Patrick Cousot and Radhia Cousot. Constructive Versions of Tarski's Fixed Point Theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.

[6] Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Forest Automata for Verification of Heap Manipulation. Technical Report FIT-TR-2011-001, FIT BUT, 2011.

[7] Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Forest Automata for Verification of Heap Manipulation. In *Proc. of CAV 2011*, volume 6806, pages 424–440. Springer Berlin Heidelberg, 2011.

[8] Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Forester: Shape analysis using tree automata. In *Proc. of 2015*, volume 9035, pages 432–435. Springer Berlin Heidelberg, 2015.

[9] Martin Hruška. Verification of Pointer Programs Based on Forest Automata, 2015.

[10] Web pages of Forester. Forester. `http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/`, 2012 [cit. 2016-01-28].