Master Thesis

Sally Chau

April 26, 2019

Contents

1 Preparation

1.1 Topic 1: Backward Confluence for Separation Logic/ Graph Grammars

1.1.1 Summary

Both Graph Grammars and Separation Logic are used to represent Heap Configurations.
Graph Grammars can be used to verify structural and relational properties for heaps in shape analysis. For the state space generation (abstraction) to terminate, we have to check whether the program state of an abstracted heap configuration has already been encountered before. Otherwise a single loop in the program typically means that we will generate new states forever. The question "has this abstract heap configuration been encountered before" is an instance of the language inclusion problem:
We ask whether the language of a newly discovered abstract heap configuration is included in the language of some previously discovered heap configuration, which is generally undecidable.

Backward confluence is beneficial for abstraction, because instead of applying all combinations of inverse derivation steps, we know that - no matter in which combination we apply inverse derivations - we end up with the same unique abstract heap configuration. Backward confluence does not exclude critical pairs of derivation rules. It rather requires that heap configurations arising from backward derivation steps using overlapping rules must be joinable, i.e., (backwards) derivable to the same heap. Hence, backward confluence speeds up abstraction.
There are classes of Graph Grammars that can be constructed to be backward confluent, however, not all grammars obtain this property.
As a fragment of SL can be translated to Graph Grammars, results on language inclusion and emptiness for Graph Grammars can be transferred to entailment and satisfiability of SL. Thus, a possible result on backward confluent Graph Grammars can also be transferred to SL.
Cyclic proofs are a semi-algorithm to decide the entailment (language inclusion problem) for SL. If we can find a cyclic proof, then we know that we found a solution. However, termination is not guaranteed.

1.1.2 Tasks

1. Define the subclass of graph grammars that can be constructed to be backward confluent.

2. Transfer the notion of backward confluence to SL; properly formulate criteria, e.g. critical pairs, for SL to see whether these conditions are "natural" in an SL setting. How do the previous findings for graph grammars transfer to SL?

3. Is it possible to design an algorithm to find cyclic proofs for backward confluent grammars/ predicates? What are the requirements?

1.1.3 Lecture on Static Program Analysis

Lecture 1: Introduction to Program Analysis

- Static program analysis: general method of automated reasoning of artefacts (requirements, design models, programs)

- Static: based on source code, not on dynamic execution

- Automated: little user intervention

- Applications

  - Optimizing compilers: improve runtime, memory efficiency
  - Software verification: verify program correctness
- Correctness = Soundness + Completeness

Lecture 2: Dataflow Analysis I (Introduction and Available Expressions/ Live Variables Analysis)

- Dataflow Analysis: describe how analysis information flows through program
  - Dependence on statement order (flow-sensitive vs. flow-insensitive analyses)
  - Direction of flow (forward vs. backward analyses)
  - Quantification over paths (may (union) vs. must (intersection) analyses)
  - Procedures (interprocedural vs intraprocedural analyses)
- Define
  - labels to locate information incl. init and final labels
  - flow relation $flow(c) \subseteq Lab \times Lab$
- Examples
  - Available Expressions Analysis: determine which complex expressions must have been computed, and not later modified, on all paths to the program point
    - Flow-sensitive
    - Forward
    - Must
  - Live Variables Analysis: determine which variables may be live at the exit from the point
    - Flow-sensitive
    - Backward
    - May

Lecture 3: Dataflow Analysis II (Order-Theoretic Foundations)

- Find similarities between analysis problems/ underlying framework
- Goal: Solve dataflow equation system by *fixpoint interation*
  1. Characterise solution of equation system as *fixpoint of a transformation*
  2. Introduce *partial order* for comparing analysis results
  3. Establish *least upper bound* as combination operator
  4. Ensure *monotonicity of transfer functions*
  5. Guarantee termination of fixpoint iteration by *ascending chain condition*
  6. Optimise fixpoint iteration by *worklist algorithm*
- Domain of analysis information usually forms a partial order where the ordering relation compares the precision of information

Definition 1.1: Partial order

A **partial order** (PO) $(D, \sqsubseteq)$ consists of a set D, called **domain**, and of a relation $\sqsubseteq \subseteq D \times D$ such that for every $d_1, d_2, d_3 \in D$,

- reflexivity: $d_1 \sqsubseteq d_1$,
- transitivity: $d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_3 \Rightarrow d_1 \sqsubseteq d_3$, and
- antisymmetry: $d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_1 \Rightarrow d_1 = d_2$

It is called **total** if, in addition, always $d_1 \sqsubseteq d_2$ or $d_2 \sqsubseteq d_1$.

- In the dataflow equation system, analysis information from several predecessors is combined by taking the least upper bound (LUB)

Definition 1.2: (Least) upper bound

Let $(D, \sqsubseteq)$ be a partial order and $S \subseteq D$.

1. An element $d \in D$ is called an **upper bound** of S if $s \sqsubseteq d$ for every $s \in S$ (notation: $S \sqsubseteq d$).
2. An upper bound d of S is called **least upper bound** (LUB) or **supremum** of S if $d \sqsubseteq d'$ for every upper bound d' of S

(notation: $d = \bigsqcup S$).

- Complete lattices ensure existence of LUB for arbitrary subsets

Definition 1.3: Complete lattice

A **complete lattice** is a partial order $(D, \sqsubseteq)$ such that all subsets of D have least upper bounds. $\bot := \bigsqcup \emptyset$ denotes the **least element** of D.

- Duality in complete lattices:

Theorem 1.1: Duality in complete lattices

$(D, \sqsubseteq)$ is a complete lattice if

- Every subset of D has a LUB
- Every subset of D has a greatest lower bound (GLB)
- it has a greatest element $\top := \sqcap \emptyset$

- Chains are generated by the approximation of the analysis information in the fixpoint iteration

Definition 1.4: Chain

Let $(D, \sqsubseteq)$ be a partial order. A subset $S \subseteq D$ is called a **chain** in D if for every $d_1, d_2 \in S$, $d_1 \sqsubseteq d_2$ or $d_2 \sqsubseteq d_1$. That is, S is a totally ordered subset of D. $(D, \sqsubseteq)$ has **finite height** if all chains are finite. Here, its **height** is $\max\{|S| | S \text{ chain in } D\} - 1$.

- Ascending Chain Condition (ACC) guarantees termination of fixpoint iteration

Definition 1.5: Ascending Chain Condition

A sequence $(d_i)_{i \in N}$ is called an **ascending chain** in D if $d_i \sqsubseteq d_{i+1}$ for each $i \in N$. A partial order $(D, \sqsubseteq)$ satisfies ACC if each ascending chain $d_0 \sqsubseteq d_1 \sqsubseteq ...$ eventually stabilizes, i.e., there exists $n \in N$ such that $d_n = d_{n+1} = ...$

- Finite height property implies ACC, but not vice versa (non-stabilising descending chains can exist)

$\Rightarrow$ Domain requirements for dataflow analysis: $(D, \sqsubseteq)$ must be a **complete lattice satisfying ACC**.

Lecture 4: Dataflow Analysis III (The Framework)

- Monotonicity of transfer functions excludes oscillating behaviour in fixpoint iteration

> **Definition 1.6: Monotonicity**
>
> Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be partial orders, and let $\phi : D \to D'$. ϕ is called **monotonic** (w.r.t. $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$) if, for every $d_1, d_2 \in D$, $d_1 \sqsubseteq d_2 \Rightarrow \phi(d_1) \sqsubseteq' \phi(d_2)$.

- Fixpoint

> **Definition 1.7: Fixpoint**
>
> Let D be some domain, $d \in D$, and $\phi : D \to D$. If $\phi(d) = d$, then d is called a **fixpoint** of ϕ.

> **Theorem 1.2: Fixpoint Theorem by Tarski and Knaster**
>
> Let $(D, \sqsubseteq)$ be a complete lattice satisfying ACC and $\phi : D \to D$ monotonic. Then $fix(\phi) := \bigsqcup\{\phi^k(\bot) | k \in N\}$ is the **least fixpoint** of ϕ where $\phi^0(d) := d$ and $\phi^(k+1)(d) := \phi(\phi^k(d))$.

> **Lemma 1.1**
>
> Let $(D, \sqsubseteq)$ be a complete lattice satisfying ACC, $S \subseteq D$ a chain, and $\phi : D \to D$ monotonic. Then $\phi(\bigsqcup S) = \bigsqcup \phi(S)$.

- Dataflow System

Definition 1.8: Dataflow System

A **dataflow system** $S = (Lab, E, F, (D, \sqsubseteq), \iota, \phi)$ consists of

- a finite set of (program) **labels** Lab
- a set of **extremal labels** $E \subseteq Lab$
- a **flow relation** $F \subseteq Lab \times Lab$
- a **complete lattice** $(D, \sqsubseteq)$ satisfying **ACC** (with LUB operator $\bigsqcup$ and least element $\bot$)
- an **extremal value** $\iota \in D$ (for the extremal labels), and
- a collection of **monotonic transfer functions** $\{\varphi_l | l \in Lab\}$ of type $\varphi_l : D \to D$

- Dataflow equation system

Definition 1.9: Dataflow equation system

Given a dataflow system $S = (Lab, E, F, (D, \sqsubseteq), \iota, \phi), Lab = \{1, ..., n\}$ (w.l.o.g)

- S determines the **equation system** (where $l \in Lab$)

$$AI_l = \begin{cases} \iota & \text{if } l \in E \\ \bigsqcup\{\varphi_{l'}(AI_{l'}) | (l, l') \in F & \text{otherwise}\} \end{cases} \tag{1}$$

- $(d_1, ..., d_n) \in D^n$ is called a **solution** if

$$d_l = \begin{cases} \iota & \text{if } l \in E \\ \bigsqcup\{\varphi_{l'}(d_{l'}) | (l, l') \in F & \text{otherwise}\} \end{cases} \tag{2}$$

- S determines the **transformation**

$$\phi_S : D^n \to D^n : (d_1, ..., d_n) \mapsto (d'_1, ..., d'_n) \tag{3}$$

where

$$d'_l := \begin{cases} \iota & \text{if } l \in E \\ \bigsqcup\{\varphi_{l'}(d_{l'}) | (l, l') \in F & \text{otherwise}\} \end{cases} \tag{4}$$

- $(d_1, ...d_n) \in D^n$ solves the equation system iff it is a fixpoint of ϕ_S
- if height of $(D, \sqsubseteq)$ is m, then the height of $(D^n, \sqsubseteq^n)$ is $m \cdot n \Rightarrow$ fixpoint iteration requires at most $m \cdot n$ steps
- solutions might not be unique: choose greatest or least solution, according to best optimisation potential

Lecture 5: Dataflow Analysis IV (Worklist Algorithm & MOP Solution

- Worklist algorithm: Efficient way to compute fixpoint, always terminates
- Meet Over all Paths - Solution (MOP-Solution): compute analysis information for block B^l by computing **least upper bound over all paths leading to** l (most precise information for l)
- Paths

Definition 1.10: Paths

Let $S = (Lab, E, F, (D, \sqsubseteq), \iota, \phi)$ be a dataflow system. For every $l \in Lab$, the set of paths up to l is given by

$$Path(l) := \{[l_1, ..., l_{k-1}] | k \geq 1, l_1 \in E, (l_i, l_{i+1}) \in F, \forall 1 \leq i < k, l_k = l\}.$$

For a path $\pi = [l_1, ..., l_{k-1}] \in Path(l)$, we define the transfer function $\varphi_\pi : D \to D$ by

$$\varphi_\pi := \varphi_{l_{k-1}} \circ ... \circ \varphi_{l_1} \circ \mathrm{id}_D$$

so that $\varphi_{[]} = \mathrm{id}_D$.

Definition 1.11: MOP solution

Let $S = (Lab, E, F, (D, \sqsubseteq), \iota, \phi)$ be a dataflow system where $Lab = \{l_1, ..., l_n\}$. The **MOP solution** for S is determined by

$$\mathrm{mop}(S) := (\mathrm{mop}(l_1), ..., \mathrm{mop}(l_n)) \in D^n$$

where, for every $l \in Lab$,

$$\text{mop}(l) := \bigsqcup\{\varphi_\pi(\iota) | \pi \in Path(l)\}.$$

- $Path(l)$ is generally infinite (loops)
- MOP solution is **undecidable** (proof based on undecidability of **Modified Post Correspondence Problem**)
- Example
  - Constant Propagation Analysis: determine, for each program point, whether a variable has a constant value whenever execution reaches that point

Lecture 6: Dataflow Analysis V (MOP vs. Fixpoint Solution

- MOP vs. Fixpoint Solution

Theorem 1.3: MOP vs. Fixpoint Solution

Let $S = (Lab, E, F, (D, \sqsubseteq), \iota, \phi)$ be a dataflow system. Then

$$mop(S) \sqsubseteq fix(\phi_S).$$

- $mop(S) \neq fix(\phi_S)$ is possible
- A sufficient condition for the coincidence of MOP and Fixpoint Solution is the distributivity of transfer functions.

Definition 1.12: Distributivity

- Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be complete lattices. Function $F : D \to D'$ is called **distributive** (w.r.t $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$) if, for every $d_1, d_2 \in D$,

$$F(d_1 \sqcup_D d_2) = F(d_1) \sqcup_{D'} F(d_2).$$

- A dataflow system $S = (Lab, E, F, (D, \sqsubseteq), \iota, \phi)$ is called **distributive** if every $\varphi_l : D \to D$ $(l \in Lab)$ is so.

Theorem 1.4

Let $S = (Lab, E, F, (D, \sqsubseteq), \iota, \phi)$ be a **distributive** dataflow system. Then

$$mop(S) = fix(\varphi_S).$$

Lecture 7: Dataflow Analysis VI (Undecidability of MOP & Non-ACC Domains

- if $(D, \sqsubseteq)$ has non-stabilising ascending chains, algorithm may not terminate $\Rightarrow$ use widening operators to enforce termination

1.1.4 Heap Abstraction Beyond Context-Freeness

Abstract

- Shape analysis is used to discover abstractions of reachable data structures in a program's heap.
- The paper develops a shape analysis to reason about relational properties of data structures.
- Hypergraphs represent the concrete and abstract domain.
- Extend context-free graph grammars to indexed graph-grammars in order to model complex data structures.
- Indexed graph grammars are used for concretization and abstraction.
- The analysis offers a high degree of automation.

Introduction

- Shape analysis aims at supporting program verification by discovering precise abstractions of reachable data structures in a program's heap.
- Therefore, we need to track detailed information about the heap configurations during computations.
- Abstractions beyond structural shape properties (e.g., balancedness) are challenging.

- In this paper: Develop shape analysis that is capable of inferring relational properties (balancedness, etc.) from a program and data structure specified by a graph grammar.
- Context-free graph grammars are not expressive enough for the analysis of relational properties.
- Thus, similar to index grammars as an extension of context-free string grammars, we introduce indexed graph grammars as an extensions of context-free graph grammars:
  - Attach an index (finite sequence of symbols) to each nonterminal: the additional information gives fine-grained control over the applicable rules.
- Indexed graph grammars offer an intuitive formalism for specifying data structures.
- Graph transformations can be applies to shape analysis:
  - Materialization: partially concretize before performing a strong update of the heap (grammar derivations)
  - Concretization: exhaustively apply derivations
  - Abstraction: exhaustively apply reverse derivations
- Subsumption between two abstract states is an instance of the language inclusion problem for graph grammars. This problem is undecidable in general.
- A fragment of indexed graph-grammars with a decidable language inclusion problem is well-suited for shape analysis.

Contributions of this paper

- Lift context-free graph grammars to indexed hyperedge replacement grammars. These are suitable for shape analysis and expressive enough to expres relational properties.
- Develop a shape analysis capable of reasoning about relational properties of data structures.
- Backward confluent grammars allow for efficient computation of abstractions. Show that the language inclusion problem is decidable for such grammars.
- Implementation of above shape analysis (Atestor).

Informal Example

- Analysis is a standard forward abstract interpretation.
- Approxiamte for each program location the set of reachable memory states.
- Use fixpoint iteration to reach results in abstract domain.
- The analysis is parameterized using an indexed hyperedge replacement grammar.
- Steps:
  - Derive abstract program semantics from concrete semantics.
  - Obtain abstraction and concretization functions.
- Challenge to show that tree is still balanced after executing example code (searchAndSwap()).
  - Show properties for arbitrary node in AVL tree.
  - Abstraction needs to be precise enough to recover that a tree remains balanced.
  - Abstraction needs to deal with destructive updates that temporarily destroy the tree shape.
- Abstract Domain
  - The abstract domain has to capture the content of the heap at each program location.
  - Sets of memory states are modeled as indexed hypergraphs including nodes, edges and indices that contain more information.
- Abstraction and Concretization
  - The set of heaps are determined by a user-supplied indexed hyperedge replacement grammar.
  - Map a hyperedge labeled with a nonterminal symbol and an index to and indexed hypergraph.
- Hypergraphs can model data structures with certain properties, e.g., the set of all balanced binary trees of height at least one with a variable n at the root.

- The full analysis explores all possible abstract executions that occur.
- Abstract semantics and concretization functions are derived automatically from the grammar and the concrete program semantics.
- Concretization corresponds to forward derivations.
- Abstraction corresponds to backward derivations.
- After a series of materialization steps, we run through respective abstraction steps (graph-based abstraction and index abstraction) according to the given grammar:
  - Materialization is used to run through the program steps and to follow the tree operations executed by the program (searchAndSwap()).
  - Abstraction is used to check if the resulting tree from the program execution still fulfills the target properties.

Preliminaries

Definition 1.13: Sets and Sequences

Given a set S,

- $\mathcal{P}(S)$ denotes the powerset of S,
- S^* is the set of all finite sequences (including the empty set ε) over S,
- $S^+ = S^* \setminus \{\varepsilon\}$,
- If $\sigma \in S^*$ and $1 \leq k \leq |\sigma|$, where $|\sigma|$ is the length of σ, the k-th element of σ is denoted by $\sigma(k)$,
- Given $\sigma, \tau \in S^*$ and $s \in S, \sigma[s \mapsto \tau]$ is the syntactic replacement of all occurrences of s in σ by τ.

Definition 1.14: Partial functions

Given a partial function $f : S \to D$,

- We write $f(s) = \perp$ if f is undefined on s.
- The restriction of f to domain $T \subseteq S$ is $f \restriction T$.

- The sequential composition of partial functions $f : S \to T, g : T \to R$ is denoted by $f \circ g$, i.e., $(f \circ g)(s) = g(f(s))$.
- For a partial order $(D, \sqsubseteq)$, $f \dot{\sqsubseteq} g$ denotes that functions $f, g : S \to D$ are ordered by pointwise application of $\sqsubseteq$.

Definition 1.15: Binary Relations

Given a binary relation $\Rightarrow \subseteq S \times S$,

- The inverse of $\Rightarrow$ is denoted by $\Leftarrow$. That is, $s \Leftarrow t$ iff $t \Rightarrow s$.
- $\not\Rightarrow s$ denotes that there exists no element $t \in S$ such that $s \Rightarrow t$.
- The reflexive and transitive closure of $\Rightarrow$ is $\Rightarrow^*$.
- $R_1 \circ R_2$ denotes the composition of binary relations R_1 and R_2, i.e., $R_1 \circ R_2 = \{(u, w) \mid \exists v : (u, v) \in R_1, (v, w) \in R_2\}$.

Definition 1.16: Context-free string grammars

Let I_N and I_T be disjoint finite sets of non-terminal and terminal symbols. A context-free string grammar (CFG) over $I = I_N \cup I_T$ is a finite set $C \subseteq I_N \times I^+$ of rules of the form $X \to \sigma$.

- Given two sequences $\sigma, \tau \in I^+$, we say that C directly derives τ from σ, written $\sigma \Rightarrow_C \tau$, if there exists a rule $(X \to \rho) \in C$, $\sigma = \sigma_1 X \sigma_2$, and $\tau = \sigma_1 \rho \sigma_2$.
- The language of a CFG C is given by the function $L_C : I^+ \to \mathcal{P}(I_T^+)$ with
$$L_C(\sigma) = \{\tau \in I_T^+ \mid \sigma \Rightarrow_C^* \tau\}.$$
- The inverse language of C is given by the function $L_C^{-1} : I^+ \to \mathcal{P}(I^+)$ that takes a string and applies inverse derivations to it as long as possible. That is,
$$L_C^{-1} = \{\tau \in I^+ \mid \sigma_C \Leftarrow^* \tau \text{ and } \tau_C \not\Leftarrow\}.$$

Concrete Semantics

- In this shape analysis, program states consist of a heap and a stack.

- The model is storeless and disregards concrete memory addresses.
- The heap consists of records with a finite number of reference fields that are collected in **Fields**. Heaps are modeled as directed edge-labeled graphs:
  - record = node
  - pointer between two records = directed edge labeled with respective field
- The stack maps program variables in **Var** to records
  - stack = special edges labeled with variables incident to nodes (= records)

Definition 1.17: Hypergraph

Let S be a set equipped with a ranking function $rank : S \to \mathbb{N}$. A hypergraph of S is a tuple $H = (V, E, lab, att)$, where

- V and E are finite sets of nodes and hyperedges, respectively,
- $lab : E \to S$ is a hyperedge labeling function, and
- $att : E \to V^*$ is an attachment function respecting the rank of hyperedge labels, i.e. for all $e \in E$, we have $rank(lab(e)) = |att(e)|$

- We consider graphs over the set $\mathbf{Var} \cup \mathbf{Fields}$.
- Variables have rank 1; fields have rank 2.
- Restrict to graphs that represent valid program states.

Definition 1.18: Heap configuration

A heap configuration (HC) is a graph H over a set containing $\mathbf{Var} \cup \mathbf{Fields}$ such that

1. every variable $x \in \mathbf{Var}$ occurs at most once in H,
2. for each field $f \in \mathbf{Fields}$, every node in H has at most one outgoing edge labeled with f, i.e. for $v \in V_H$ there is at most one $e \in E_H$ with $att_H(e)(1) = v$ and $lab_H(e) = f$,
3. there is a unique node $v_{\texttt{null}}$ without outgoing edges labeled with

fields.

The set of all heap configurations is denoted by **HC**.

- Define a small heap-manipulating programming language: `Progs`-program.

Definition 1.19: `Progs`-programs

Let $x \in$ **Var** be a variable and $f \in$ **Fields** be a field. Then the syntax of programs `Progs` (P), Boolean expression `BExp` (B) and pointer expressions `PExp` (Ptr) is defined by the following context-free grammar:

- $P ::= \; x = Ptr \mid x.f = Ptr \mid \texttt{new}(x) \mid P; P \mid \texttt{noop} \mid$

  $\texttt{if} \; (B) \; \{P\} \; \texttt{else} \; \{P\} \mid \texttt{while} \; (B) \; \{P\}$

- $B ::= \; Ptr = Ptr \mid B \wedge B \mid \neg B$

- $Ptr ::= \; \texttt{null} \mid x \mid x.f$

- The semantics of `Progs`-programs is given by a function

  $$\mathcal{C}[\![.]\!] : \texttt{Progs} \to \mathbf{HC} \to \mathbf{HC},$$

  that takes a program P and a program state (a heap configuration H) and yields a (new, updated) heap configuration capturing the effect of executing P on H (see page 10 for details on semantic functions).

Indexed Hyperedge Replacement Grammars

- Add additional edges as placeholders for possibly infinite sets of graphs.
- Theses edges are defined by hyperedge replacement grammars (HRG) lifted to indexed hyperedge replacement grammars for more expressiveness.
- Extend graphs by indices.

Definition 1.20: Indexed hypergraph

Let N and T be disjoint finite sets of nonterminals and terminals that are equipped with a ranking function $rank : (N \cup T) \to \mathbb{N}$ and let I a finite non-empty set of index symbols.
An indexed hypergraph over the sets $N \cup T$ and I is a tuple $H = (V, E, lab, att, ind, ext)$, where

- (V, E, lab, att) is a hypergraph over $N \cup T$ according to Definition 1.17,
- $ind : E \to I^+$ assigns an index to each edge in E, and
- $ext \in V^*$ is a (possibly empty) repetition-free sequence of external nodes.

Definition 1.21: Indexed hyperedge replacement grammar (IG)

Let v be a dedicated index variable that is not contained in the set of index symbols I. An indexed hyperedge replacement grammar (IG) is a finite set of rules G of the form $X, \sigma \to H$ mapping a nonterminal $X \in N$ and an index $\sigma \in I^*(I \cup \{v\})$ to an indexed graph H over $N \cup T$ and $I \cup \{v\}$ such that $rank(X) = |ext_H|$. Moreover, if σ does not contain the variable v then H does not contain v either, i.e. $ind_H(E_H) \subseteq I^+$.

- IG derivation replaces an edge e that is labeled with a nonterminal by a finite graph K.
- External nodes of K identify each node attached to the edge e with a note of K.

Definition 1.22: External Nodes

Let H, K be indexed graphs over $N \cup T$ and I with pairwise disjoint sets of nodes and edges. Moreover, let $e \in E_H$ be an edge such that $rank(lab_{E_H}(e)) = |ext_K|$. Then the replacement of e in H by K is given by $H[e \mapsto K] = (V, E, att, lab, ind, ext)$, where

- $V = V_H \cup (V_K \setminus ext_K)$
- $E = (E_H \setminus \{e\}9 \cup E_K$
- $lab = (lab_H \upharpoonright E') \cup lab_K$
- $ind = (ind_H \upharpoonright E') \cup ind_K$
- $att = (att_H \upharpoonright E') \cup (att_K \circ mod)$
- $ext = ext_H$,

where *mod* replaces each external node by the corresponding node attached to e, i.e.

$mod = \{ext_K(k) \mapsto att_H(e)(k) | 1 \leq k \leq |ext_K|\} \cup \{v \mapsto v | v \in V \setminus ext_K\}$.

Definition 1.23

Given a set $M \subseteq T \cup N$, we write E_H^M to refer to all edges of H that are labeled with a symbol in M, i.e. $E_H^M = \{e \in E_H | lab_H(e) \in M\}$.
As for strings, we write $H[v \mapsto \rho]$ to replace all occurences of v in (the index function of) H by ρ.

Definition 1.24: Index derivations

Let G be an I and H, K be indexed hypergraphs over $N \cup T$ and I. Then G directly derives K from H, written $H \Rightarrow_G K$, iff either

- there exists a rule $(X, \sigma \to R) \in G$ and an edge $e \in E_H^{\{X\}}$ such that $ind_H(e) = \sigma$ and K is isomorphic to $H[e \mapsto R]$ or
- there exists a rule $(X, \sigma v \to R) \in G$, an edge $e \in E_H^{\{H\}}$, and a sequence $\rho \in I^+$ such that $ind_H(e) = \sigma\rho$ and K is isomorphic to $H[e \mapsto R[v \mapsto \rho]]$.

Definition 1.25: Isomorphic indexed graphs

Two indexed graphs H and K are isomorphic, written $H \cong K$, iff there exist bijective functions $f : V_H \to V_K$, $g : E_H \to E_K$ such that

- for each $e \in E_H$, $lab_H(e) = lab_K(g(e))$ and $ind_H(e) = ind_K(g(e))$,
- for each $e \in E_H$, $f(att_H(e)) = att_K(g(e))$, and $f(ext_H) = ext_K$.

- We assume that all rules of an IG G are increasing, i.e. for each rule $(X, \sigma \to H) \in G$ it holds that

$$|V_H| + |E_H| > rank(X) + 1.$$

- The language of an IG and an indexed graph H is the set of all graphs that can be derived from H and that contain terminal edge labels only.

- The inverse language of H is obtained by exhaustively applying inverse derivations to H.

Definition 1.26: Language and inverse language of IGs

The language L_G and the inverse language L_G^{-1} of IG G are given by the following functions mapping indexed graphs to sets of indexed graphs:

$$L_G(H) = \{K | H \Rightarrow_G^* K \text{ and } E_K = E_K^T\}$$

and

$$L_G^{-1} = \{K | H \Leftarrow_G^* K \text{ and } K \not\Leftarrow_G\}.$$

Theorem 1.5: Properties of IGs

Let G be and IG and H be an indexed graph over $N \cup T$. Then

1. $H \Rightarrow_G^* K$ implies $L_G(K) \subseteq L_G(H)$.
2. $L_G(H) = \begin{cases} \{H\}, & \text{if } \neg\exists e \in E_H : lab_H(e) \in N. \\ \bigcup_{H \Rightarrow_G K} L_G(K), & \text{otherwise.} \end{cases}$
3. It is decidable whether $L_G(H) = \emptyset$ holds.
4. The inverse language $L_G^{-1}(H)$ is non-empty and finite.

Abstract Semantics

- This analysis is a forward abstract interpretation. The indexed graph grammar overapproximates the set of reachable heap configurations for each program location.

Definition 1.27: Indexed Heap Configuration (IHC)

An indexed hypergraph $H = (V, E, lab, att, ind, ext)$ over $\mathbf{Types} = \mathbf{Var} \cup \mathbf{Fields} \cup N$, where N is a finite set of nonterminals, and I is an indexed heap configuration (IHC) if

- its underlying hypergraph (V, E, lab, att) is a heap configuration,
- the first external node corresponds to `null`, i.e. $v_{\texttt{null}} = ext(1)$, and

- for each $e \in E^N$, the first attached node id $v_{\texttt{null}}$, i.e. $att(e)(1) = v_{\texttt{null}}$.

The set of all indexed heap configurations is denoted by **IHC**.

Definition 1.28: Our Shape Analysis

- Concrete domain: $(\mathbf{Con} = \mathcal{P}(\underbrace{L_G(\mathbf{IHC})}_{\text{concrete IHCs}}), \subseteq)$
- Concretization: $\gamma = L_G$
- Abstract domain: $(\mathbf{Abs} = \mathcal{P}(\underbrace{L_G^{-1}(\mathbf{IHC})}_{\text{fully abstract IHCs}}), \sqsubseteq)$
  - $\sqsubseteq$ is given by $H \sqsubseteq K$ iff $\gamma(H) \subseteq \gamma(K)$.
- Abstraction: $\alpha = L_G^{-1}$

- The analysis performs a fixed-point iteration of the abstract semantics that overapproximates the concrete semantics.
- Abstract semantics consists of three phases:
  - materialization
  - execution of concrete semantics
  - canonicalization
- Abstract semantics is defined by the function

$$\mathcal{A}[\![.]\!] : \texttt{Progs} \to \mathbf{Abs} \to \mathbf{Abs}$$

- The indexed graph grammar G influences the abstract semantics through materialization and canonicalization
- Materialization
  - Lift concrete semantics $\mathcal{C}[\![.]\!]$ from **HC** to **IHC**
  - Partially concretizes an IHC such that the concrete semantics is applicable

$$materialize[\![.]\!] : \texttt{Progs} \to \mathbf{IHC} \to \mathcal{P}_{\text{finite}}(\mathbf{IHC})$$

– Intuitively: applies derivations $\Rightarrow_G$ until the concrete semantics can be applied; maps IHC to finite sets of IHCs.

Definition 1.29: *materialize* $[\![.]\!]$

Let P be either a `Progs`-program of the form $x = Ptr$, $x.f = Ptr$, `new`(x), `noop`, or a Boolean expression. Then

$$\gamma \circ \mathcal{C}[\![\mathcal{P}]\!] \dot{\subseteq} \mathit{materialize}\ [\![\mathcal{P}]\!] \circ \mathcal{C}[\![\mathcal{P}]\!] \circ \gamma.$$

- Canonicalization
  - "Conversely" to materialization
  - Takes a partially concretized program state and computes an abstract program state (after execution of the concrete semantics).

$$\mathit{canonicalize}[\![.]\!] : \texttt{Progs} \rightarrow \mathbf{IHC} \rightarrow \mathbf{Abs}$$

Definition 1.30: *canonicalize* $[\![.]\!]$

For all $P \in \texttt{Progs} : \gamma \dot{\subseteq} \mathit{canonicalize}\ [\![P]\!] \circ \gamma$.

- The quality of the analysis depends on the input grammar.
- The termination of the analysis needs to be ensured by widening, e.g., by fixing a maximal IHC size.
- Soundness of analysis: Show that the abstract semantics $\mathcal{A}[\![.]\!]$ computes an overapproximation of the concrete semantics $\mathcal{C}[\![.]\!]$.

Theorem 1.6: Soundness

For all $P \in \texttt{Progs} : \gamma \circ \mathcal{C}[\![P]\!] \dot{\subseteq} \mathcal{A}[\![P]\!] \circ \gamma$.

Theorem 1.7: Local Reasoning

Let $P \in \texttt{Progs}$ and H, R be IHCs. Let $Mod[\![P]\!](H)$ denote the set of all nodes and edges that are added or deleted when running P on H. Moreover, $H \cup R$ denotes the componentwise union of two IHCs.
Then

$$\mathcal{C}[\![P]\!](H) = K$$

and

$$Mod[\![P]\!](H) \cap (V_R \cup E_R) = \emptyset$$

implies

$$\mathcal{C}[\![P]\!](H \cup R) = K \cup R.$$

Backward Confluent IGs

- Canonicalization requires repeated computation of inverse language (exhaustively applying inverse derivations). This requires to solve the isomorphic subgraph-problem, which is NP-complete.
- Fixed point computation includes language inclusion problem, which is undecidable.
- Study indexed graph grammars that have unique canonicalization (there exists only one unique inverse language) and for which the inclusion problem is decidable.

Definition 1.31: Backward Confluent IGs

An IG G is backward confluent iff for all IHCs H the inverse language $L_G^{-1}(H)$ is a singleton set, i.e. $|L_G^{-1}(H)| = 1$.

- Unique inverse language property
  - A critical pair is an IHC that is obtained by gluing two IHCs that occur on the right-hand side of IG rules together and that overlap not only in their external nodes.
  - An IG is backward confluent iff all of its critical pairs are joinable: Applying the two inverse derivations result in the same IHC.
  - Most (not all!) IGs can be transformed into backward confluent grammars by repeatedly joining critical pairs (instance of Knuth-Bendix completion algorithm).

Theorem 1.8: Language of Backward Confluent IGs

There exist languages of IHCs that can be generated by an IG, but not by any backward confluent IG.

- Decidable inclusion problem property
  - Two IHCs to which no inverse derivation is applicable are either isomorphic or have disjoint languages.

Theorem 1.9:

Let G be a backward confluent IG. Moreover, let H, $K \in \mathbf{IHC}$ such that $H \not\Leftarrow_G$ and $K \not\Leftarrow_G$. Then

$$L_G(H) \cap L_G(K) \neq \emptyset \text{ iff } H \text{ is isomorphic to } K.$$

Theorem 1.10:

Let G be a backward confluent IG. Moreover, let H, $K \in \mathbf{IHC}$ such that $K \not\Leftarrow_G$. Then it is decidable whether $L_G(H) \subseteq L_G(K)$ holds.

- Backward confluent IGs yield a unique abstraction.
- The abstract and concrete domain form a Galois connection.
- This shape analysis is applicable without backward confluent IGs, but will be computationally more expensive and less precise.

Global Index Abstraction

- Above abstraction till requires to memorize the indices. Abstract from this by index abstraction (keep track of the differences between indices instead).
- Remove common suffic from all indices and replace by a placeholder.
- Index abstraction is formalized in a right-linear CFG.

Definition 1.32: Well-formed index

Let $I = I_N \cup I_T$ be a finite set of index symbols with a set of nonterminals I_N and a set of terminals I_T including the end-of-index symbol z.
An index $\sigma \in I^+$ is well-formed if $\sigma \in (I_T \setminus \{z\})^*(I_N \cup \{z\}))$. This means that a well-formed index always ends with a nonterminal or the end-of-index symbol z.
An IHC is well-formed if all of its indices are.

- All indices ending with the same nonterminal of an IHC are modified simultaneously. This requires global derivations/ languages.
- Global derivation affect indices only, no edge replacement is involved.

Definition 1.33: Global Derivation

Let H, $K \in \mathbf{IHC}$. A CFG C globally derives K from H, written $H \rightrightarrows_C K$, iff there exists a rule $(X \to \tau) \in C$ such that

$$ind_H(E_N^N) \subseteq I_T^* I_N$$

and

$$K \text{ is isomorphic to } H[X \mapsto \tau].$$

Definition 1.34: (Inverse) Global Language

The global language and the inverse global language of a right-linear CFG C over I are given by

$$GL_C : \mathbf{IHC} \to \mathcal{P}(\mathbf{IHC}), H \mapsto \{K | H \rightrightarrows_C^* K \text{ and } ind_K(E_K^N) \subseteq I_T^+\}$$

$$GL_C^{-1} : \mathbf{IHC} \to \mathcal{P}(\mathbf{IHC}), H \mapsto \{K | H \leftrightarrows_C^* K \text{ and } K \not\leftrightarrows_C\}$$

Lemma 1.2

If H is well-formed and $H \rightrightarrows_C^* K$ or $H \Rightarrow_G^* K$, then K is well-formed.

- Global derivations have same properties as IG derivations in order to ensure soundness and termination of abstraction.

Theorem 1.11: Properties of Global Derivations

Let C be a CFG and H, $K \in \mathbf{IHC}$. Then

1. $H \rightrightarrows_C^*$ implies $GL_C(K) \subseteq GL_C(H)$.
2. $GL_C(H) = \begin{cases} \{H\} & \text{if } ind_H(E_H^N) \subseteq I_T^* z \\ \bigcup_{H \rightrightarrows_C K} GL_C(K) & \text{otherwise.} \end{cases}$
3. If C contains a single nonterminal, i.e. $|I_N| = 1$, it is decidable

whether $GL_C(H) = \emptyset$ holds.

4. If C contains no rule of the form $X \to Y$, where $X, Y \in I_N$, then the global inverse language $GL_C^{-1}(H)$ is non-empty and finite.

- The relation $(\Rightarrow_G \cup \rightrightarrows_C)^*$ combines global derivations and IG derivations.
- Global derivations and IG derivations are orthogonal to each other.

Theorem 1.12: Orthogonality of Global and IG derivations

$$H(\Rightarrow_G \circ \rightrightarrows_C)K \text{ implies } H(\rightrightarrows_C \circ \Rightarrow_G^*)K$$

and

$$H(\Rightarrow_G \cup \rightrightarrows_C)^* K \text{ iff } H(\rightrightarrows_C^* \circ \Rightarrow_G^*)K$$

- Materialization: apply all possible global derivations and then apply IG derivations
- Abstraction: apply inverse IG derivations and then apply inverse global derivations
- Thus, only relevant relationships between indices are kept.
- Materialization and canonicalization now formally depend on user-provided CFG C.

Definition 1.35: Our Refined Analysis

- Concrete domain: $(\mathbf{Con} = \mathcal{P}(\underbrace{L_G(GL_C(\mathbf{IHC}))}_{\text{concrete IHCs}}), \subseteq)$
- Concretization: $\gamma = GL_C \circ L_G$
- Abstract domain: $(\mathbf{Abs} = \mathcal{P}(\underbrace{GL_C^{-1}(L_G^{-1}(\mathbf{IHC}))}_{\text{fully abstract IHCs}}), \sqsubseteq)$
  - $\sqsubseteq$ is given by $H \sqsubseteq K$ iff $\gamma(H) \subseteq \gamma(K)$.
- Abstraction: $\alpha = L_G^{-1} \circ GL_C^{-1}$

- Analysis is sound and decidable (for backward confluent grammars).

Theorem 1.13: Soundness

The refined analysis from above is sound. That is,

$$\forall P \in \texttt{Progs} : \gamma \circ \mathcal{C}[\![P]\!] \dot{\subseteq} \mathcal{A}[\![P]\!] \circ \gamma.$$

Theorem 1.14: Decidability

Let C and G be a backward confluent CFG and IG, respectively. Moreover, let H, K be well-formed IHCs such that $K \not\leftleftharpoons_C$ and $K \not\leftleftharpoons_G$. Then is it decidable whether $H \sqsubseteq K$ holds.

Implementation

- The above analysis has been implemented in the ATTESTOR Framework.

1.1.5 Static Analysis of Pointer Programs

Grammar-Based Abstraction of Data Structures

- Heaps can be of unboundedly large size. Thus, we model the heap by abstract parts using nonterminal edges.
- Nonterminal edges are defined by hyperedge replacement grammars.
- Separation logic is an extension of Hoare Logic.

Definition 1.36: Hyperedge Replacement Grammar (HRG)

A hyperedge replacement grammar G over an alphabet Σ, containing a set of nonterminals N, is a finite set of production rules of the form $A \to H$ where $A \in N, H \in HC_{\text{Sel} \cup N}$, and $|ext_H| = \text{rank}(A)$. The set of all HRGs over Σ is denoted by HRG_Σ.
By G^A we denote the set of all rules $A \to H \in G$ and define $\bar{G^A} := G \setminus G^A$. In the following we call the hypergraph H of rule $A \to H$ a rule graph.

Definition 1.37: Hyperedge Replacement

Let $H, K \in HG_\Sigma$ and $e \in E_H$ a nonterminal edge with $\text{rank}(e) = |ext_K|$. W.l.o.g. let $V_H \cap V_K = E_H \cap E_K = \emptyset$. The hyperedge replacement of e by K, denoted $H[K/e]$, is the hypergraph $H' \in HG_\Sigma$ defined by

- $V_{H'} = V_H \cup (V_K \setminus [ext_K])$
- $lab_{H'} = (lab_H \upharpoonright (E_H \setminus \{e\})) \cup lab_K$
- $att_{H'} = att_H \upharpoonright (E_H \setminus \{e\}) \cup (mod \circ att_K)$
- $E_{H'} = (E_H \setminus \{e\}) \cup E_K$
- $ext_{H'} = ext_H$

where

$$mod = \text{id}_{v_{H'}} \cup \{ext_K(1) \mapsto att_H(e)(1), \ldots, ext_K(\text{rank}(e)) \mapsto att_H(e)(\text{rank}(e))\}.$$

Definition 1.38: HRG Derivation

Let $G \in HRG_\Sigma, H, H' \in HG_\Sigma, p = A \to K \in G$ and $e \in E_H$ with $lab_H(e) = A$. H derives H' by p, denoted $H \Rightarrow_{e,p} H'$ iff H' is isomorphic to $H[K/e]$. Let $H \Rightarrow_G H'$ if $H \Rightarrow e, pH'$ for some $e \in E_H$ and $p \in G$. If G is clear from the context, we write $H \Rightarrow H'$ instead. Further, we let $\Rightarrow^*$ denote the reflexive and transitive closure of $\Rightarrow$.

Definition 1.39: Language of an HRG

The language generated from $H \in HG_\Sigma$ with respect to $G \in HRG_\Sigma$ is defined by

$$\mathcal{L}_G(H) = \{H' \in HG_{PVar \cup Sel} | H \Rightarrow^* H'\}.$$

- All HGs that can be derived from a nonterminal A, is refered to as the language of nonterminal A (the language generated from the handle of A).
- The handle of A is a hypergraph consisting of a single hyperedge labelled A attached to external vertices only.

Definition 1.40: Handle

Given $A \in N$ with $\text{rank}(A) = n$, an A-handle is the hypergraph

$$A^\bullet = (\{v_1, \ldots, v_n\}, \{e\}, [e \mapsto v_1 \ldots v_n], [e \mapsto A], v_1 \ldots v_n) \in HG_\Sigma.$$

The language generated from nonterminal A is given by $\mathcal{L}(A^\bullet)$.

Abstraction and Concretisation

- HRGs define sets of graphs derivable from a given start graph.
- Each nonterminal defines a class of graphs.
- Use this concept to define abstractions of heap configurations.
- Concretization: forward rule application of HRG; yields the language derivable from the abstract heap configuration.

Definition 1.41: Concretisation

Given an HRG G, we call $H' \in HC_{PVar \cup Sel}$ a concretisation of $H \in HC_\Sigma$ iff $H \Rightarrow^* H'$. A concretisation replaces all nonterminal edges until a concrete heap configuration is reached.

Definition 1.42: Concretisation Function γ

Given an HRG G, the concretisation function $\gamma_G : HC_\Sigma \to 2^{HC_{PVar \cup Sel}}$ maps an HC H to its concretisations, i.e. $\gamma_G(H) := \mathcal{L}_G(H)$.

- γ does not result in a loss of precision.

Theorem 1.15: γ precision

For $G \in HRG_\Sigma, H \in HG_\Sigma, e \in E_H, A \in N$ with $A = lab(e)$:

$$\mathcal{L}_G(H) = \bigcup_{A \to K \in G} \mathcal{L}_G(H[K/e]).$$

- Abstraction: backward rule application; complementary operation to concretisation.

Definition 1.43: Abstraction

We call $H \in HC_\Sigma$ an abstraction of $H' \in HC_\Sigma$ if $H \Rightarrow^* H'$ and there exists no $K \in HC_\Sigma$ such that $K \Rightarrow^+ H$.

Definition 1.44: Abstraction Function α

Given an HRG G, the abstraction function $\alpha_G : HC_\Sigma \to 2^{HC_\Sigma}$ maps an HC H' to its maximal abstractions, i.e. $\alpha_G(H') := \{h \in HC_\Sigma | H \Rightarrow^* H' \nexists K \in HC_\Sigma : K \Rightarrow^+ H\}$.

- α is sound and yields an overapproximation of the language induced by the HRG.
- The resulting heap configuration are at most as precise as the original one.
- α can yield several graphs.

Lemma 1.3

For $G \in DSG_\Sigma$ and $H, H' \in HC_\Sigma, H \Rightarrow^* H'$ implies $\mathcal{L}(H') \subseteq \mathcal{L}(H)$.

Data Structure Abstraction

- HRGs are not sufficient to describe data structures as they might comprise hypergraphs that do not depict heaps.
- Ensure heap configuration properties.
- DSGs only derive HCs and do no abstract information about program variables.

Definition 1.45: Data Structure Grammar (DSG)

$G \in HRG_\Sigma$ is a data structure grammar over Σ if for all $A \in N$: $\mathcal{L}(A^\bullet) \subseteq HC_{Sel}$. Let DSG_Σ denote the set of all DSGs over Σ.

Theorem 1.16: DSG Decidability

It is decidable whether an HRG is a DSG.

Datastructure Normal Form

- Supported datastructures using DSGs are such with bounded tree width.
- Interleaved application of concretisation and abstraction steps can lead to invalid heap configurations.
- Introduce Datastructure Normal Form to preserve heap properties:
  - Productivity
  - Typedness
- Productivity
  - $\mathcal{L}(\mathcal{A}^\bullet) \neq \emptyset, \forall \mathcal{A} \in \mathcal{N}$.
  - In other words, a non-terminal A is unproductive if no rule can be applied so that A can be derived to a concrete hypergraph.
  - Productivity can be achieved by removing non-productive non-terminals and corresponding production rules without changing the language of G.

Definition 1.46: Productivity

The set of productive NTs is the least set such that for each element A there exists a production rule $A \to H$ such that either $H \in HC_{Sel}$ or all element of $\{B \in N | \exists e \in E_H : lab(e) = B\}$ are productive. A $G \in HRG_\Sigma$ is productive if all of its NTs are productive.

- Typedness
  - Impose typing restrictions.
  - Only outgoing edges at external nodes can by manipulated from outside the rule graph.
  - An HRG is typed if every external vertex has a well-defined type.

Definition 1.47: Outgoing edges

Let $H \in HC_{PVar \cup Sel}$ and $v \in V_H$. The set of outgoing edges at vertex v in H is defined as

$$out(v) = \{e \in E_H | att(e)(1) = v\}.$$

Definition 1.48: Typedness

$G \in DSG_\Sigma$ is typed of for all $A \in N$ and $i \in [1, \text{rank}(A)]$, there eixtsts a set $type(A, i) \subseteq Sel$ such that for every $H \in \mathcal{L}(A^\bullet)$: $type(A, i) = lab_H(out_H(ext_H(i)))$.

Theorem 1.17: Typedness

It is decidable whether an HRG is typed. For every DSG an equivalent typed DSG can be constructed.

Definition 1.49: Data Structure Normal Form (DSNF)

$G \in DSG_\Sigma$ is in Data Structure Normal Form if it is productive and typed.

Theorem 1.18

For every DSG Data Structure Normal Form is establishable.

Canonical Abstraction

- A singleton set of abstraction is highly desirable since it leads to a smaller state space.
- The Knuth-Bendix completion algorithm establishes confluence of rewrite systems (semi-decision algorithm that establishes backward confluence of HRGs by analysing critical pairs and deriving new production rules that resolve the non-confluence for the critical pair).
- Currently not aware of a construction that establishes backward confluence for any HRG (p.53)

- "Highly desirable for future research to derive a class of DSGs, which can automatically be transformed into backward confluent ones." (p.54)

Definition 1.50: Backward Confluence

A DSG G is backward confluent iff $\alpha_G(H)$ is a singleton set for every $H \in HC_\Sigma$.

Theorem 1.19: Backward Confluence

It is decidable whether an HRG G is backward confluent.

1.1.6 Data Structure Grammars & Separation Logic

- Investigate the relationship between DSGs (datastructure grammar) and SL (separation logic).
- There exists a comprehensive fragment of SL that corresponds to DSGs.
- We provide a two-way translation from one formalism into the other: allows for correspondence results.
- Use translation for obtain decidability results for SL entailment problem and the language inclusion problem.

Separation Logic with Recursive Predicates

- In SL, a heap is a set of locations connected via references.

Definition 1.51: Heap

Let $HLoc_{=}\mathbb{N}$ be the set of heap locations. Let $Elem := HLoc \cup \{\texttt{null}\}$. A heap is a finite (partial) mapping $h : HLoc \rightharpoonup Elem$. The set of all heaps is denoted be $Heap$.

Definition 1.52: Stack

Let Var be a set of variables. A stack is a (partial) mapping $stk : Var \rightharpoonup Elem$. We denote the set of all stacks by $Stack$.

- A heap contains objects which has references to other objects represented by a finite set Sel of selectors.
- References are located in successive locations: reserve $|Sel|$ successive locations for each object.
- $\mathrm{cn}(s), s \in Sel$ denoted the ordinal selector s, where $0 \leq \mathrm{cn}(d) < |Sel|$: Under stack stk, the location of selector s of an object referred to by x is $stk(x) + \mathrm{cn}(s)$.
- A stack $stk \in Stack$ is safe if $img(stk) \subseteq \{1, |Sel|+1, 2|Sel|+1, \dots\} \cup \{\texttt{null}\}$: in safe stacks program variables only refer to objects.

Definition 1.53: Syntax of SL

Let $Pred$ be a set of predicate names, each associated with an arity. The syntax of SL is given by:

$E ::= x \mid \texttt{null}$

$P ::= y = z \mid y \neq z \mid P \wedge P$ (pure formulae)

$F ::= \mathbf{emp} \mid x.s \mapsto E \mid F * F \mid \exists x : F \mid \sigma(E_1, \dots, E_n)$ (heap formulae)

$S ::= F \mid S \vee S \mid P \wedge S$ (SL formulae)

where $x, y, z \in Var, s \in Sel$ and $\sigma \in Pred$ of arity n. A heap formula of the form $x.s \mapsto E$ is called a points-to assertion. SLF denotes the set of all SL formulae.

- Given $\phi \in SLF$, the set $Var(\phi)(FV(\phi))$ collects all (free) variables of ϕ.
- If $FV(\phi) = \emptyset$, then ϕ is called closed.
- $Atomic(\phi)$ denotes the set of all atomic subformulae of ϕ (those of the form $x = y, \mathbf{emp}, x.s \mapsto E, \sigma(E, \dots, E)$).
- The predicate calls in ϕ are given by $pred(\phi) := \{\sigma(x_1, \dots, x_n) \in Atomic(\phi) | \sigma \in \mathrm{Pred}, x_1, \dots, x_n \in Var \cup \{\texttt{null}\}\}$.
- If $pred(\phi) = \emptyset$, then ϕ is called primitive.

Definition 1.54: Environment

A predicate definition for $\sigma \in \text{Pred}$ with arity n is of the form $\sigma(x_1, \ldots, x_n) := \sigma_1 \vee \cdots \vee \sigma_m$, where $m, n \in \mathbb{N}, x_1, \ldots, x_n \in Var$ are pairwise distinct, $\sigma_1, \ldots, \sigma_m$ are heap formulae and $FV(\sigma_j) = \{x_1, \ldots x_n\}$ for each $j \in [1, m]$. We call $\sigma_1 \vee \cdots \vee \sigma_m$ the body of σ and $\sigma(x_1, \ldots, x_n)$ a predicate call. A finite set of definitions with distinct predicates is called an environment. The set of all environments is denoted by Env.

1.2 Topic 2: Hierarchical Model Checking for LTL

1.2.1 Summary

LTL model-checking is used to verify or express properties that should be satisfied by a system in the infinite run. Satisfyability checking of LTL formula is equivalent to the language-emptiness problem of ω-regular languages (languages of infinite words) which are recognized by Büchi-automata. Furthermore, the models used in model-checking can be extended to Recursive State Machines (RSMs) which allow the notion of recursion. Thus, RSMs can model the control flow in programming languages with recursive procedure calls. Hence, we can use RSMs and LTL formulae to guarantee or verify certain properties in pograms that use recursion.

1.2.2 Tasks

1. Is it possible to model Attestor's state spaces as an RSM?
2. Implement a hierarchical model checker for the state spaces generated by Attestor.
3. Attestor generates contracts which are pairs of pre- and post-conditions. How do we need to implement the contracts in state spaces? ("Inwiefern müssen wir die eigentlichen state spaces vorhalten?")
4. How do instances of counterexamples look like in the context of hierarchical model-checking?
5. If we find a counterexample, can we automatically show that it is realistic or that it is a spurious counterexample created by too much abstraction? How to extract new predicates from spurious counterexample?

1.2.3 Model-Checking

- Model-Checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state) in that model.
- A model can be a transition-system TS, similar to an automaton.
- Model-Checking is used to expose potential design errors.
- For example in a program, Model-Checking can be used to verify if all runs of a program satisfy certain properties (for different states).

1.2.4 Linear-Time Temporal Logic (LTL)

- Specification language for expressing correctness requirements of reactive systems.
- Given an abstract model M of a reactive system and an LTL formula ϕ, LTL model-checking asks whether an infinite computation of M satisfying ϕ exists.
- LTL extends propositional or predicate logic with the possibility to express linear time ordering.
- For example, something will eventually happen or something will happen once and will stay this way forever in the future.
- LTL cannot specify exact times or durations, but rather the relative order of events.
- The time domain is discrete: the current state refers to "now", the next state or transition to the next state refers to the "next" time unit.
- Syntax: additionally to atomic propositions and Boolean connectors, we have two temporal modalities: $\bigcirc$ next and $\bigcup$ until.
- Semantics: LTL formulae represent properties of paths: a path can either fulfill an LTL formula or not.

1.2.5 Büchi-Automata

- Büchi-Automaton is a non-deterministic finite state-automaton which takes infinite words as input.

- Satisfyability checking of an LTL formula is equivalent to emptiness problem of ω-regular languages (Büchi-Automata).
- Nondeterministic Büchi-Automata recognize ω-languages. An NBA recognizes an infinite word σ if there exists a run $q_0, q_1, \ldots$ for σ where $q_0 \in Q_0$ and $q_i \in F$ for infinitely many $i \in \mathbb{N}$.
- ω-regular languages
  - Σ^ω denotes the set of all infinite words over Σ.
  - Any subset of Σ^ω is called a language of infinite words or ω-language.
  - The operator ω indicated infinite repetition.
- NBAs are expressive as ω-regular languages.
- An NBA $\mathcal{A}$ is non-empty if there exists a reachable accepting state q that belongs to a cycle in $\mathcal{A}$. (can be solved in time $\mathcal{O}(|\mathcal{A}|)$)
- NBAs are more expressive than DBAs.
- GNBAs (generalized non-deterministic Büchli-Automata) accept a word σ if all accepting states from a set of accepting states $\mathcal{F}$ are visited infinitely many times.
- GNBAs can be translated into NBAs and vice versa.

1.2.6 Recursive State Machines

- Model is a finite state machine with vertices as system states and edges as system transitions.
- In Recursive State Machines (RSMs) vertices can be states or invocations of other state machines (recursion).
- RSMs can model control flow in programming languages with recursive procedure calls.
- RSMs consist of a set of component state machines where each component state machines consists of
  - a set of node and boxes
  - a labeling that assigns to every box an index of one of the component machines

  - a set of entry and exit nodes: control interface of a component by which it can communicate with other components.
  - a transition relation

- Global state-space is infinite due to recursion.
- Use RSMs to verify reachability and cycle detection and LTL properties.
- Since hierarchical state machines are a special case of RSMs, some results can be transferred/ reduced to RSMs.
  - LTL model checking, reachability and cycle detection problems for single-exit RSMs can be solved in time linear in the size of the RSM.
  - LTL model checking, reachability and cycle detection problems for multiple-exit RSMs can be solved in time cubic in the size of the RSM.
  - LTL model checking, reachability and cycle detection problems for single-entry multiple-exit RSMs can be solved in time linear in the size of the RSM.
- Given a recursive automaton, verify any properties in LTL (reachability, emptiness).
- Reachability:
  - **Step 1:** The rules and the AND-OR graph constructtion.
  - Compute for each component A_i of the RSM a predicate relation $R_i(x, y)$: there is a path in T_A from $\langle x \rangle$ to $\langle y \rangle$. Establish rules on reachability inside a system.
  - Evaluation of Datalog program corresponds to reachability analysis of corresponding AND-OR graph $G_A = (V, E, Start)$. Reachability in AND-OR graphs can be computed in linear time. Thus, we know reachability amoing its entry and exit nodes for each component.
  - **Step 2:** The augmented graph H_A.
  - Determine the set of nodes reachable from the initial set in a global manner. Build graph H_A with set of vertices as union of all sets of vertices of all the components, set of edges as union of all sets of edges of all the components. Add edges from entry to

exit vertices for according boxes and components, and add edges between each entry vertex of a box to the corresponding entry node of the component to which the box is mapped.

  - Use DFS to search for all reachable nodes in H_A.
  - The two steps caan be carried out simultaneously in practice.

- Language Emptiness:
  - Compute predicate $Z_i(x, y)$ for each component: true, if there is a path from $\langle x \rangle$ to $\langle y \rangle$ that passes through an accept state.
  - Emptiness detection boils down to cycle detection in graphs in time linear in the size of the RSM.

- Model Checking of LTL Properties and Büchi Automata:
  - Given an LTL formula, the the model checking procedure consists of
    * building a finite-state Büchi automaton $A_{\neg\phi}$ that accepts exactly all the infinite words satisfying the formula $\neg\phi$
    * computing the product of $A_{\neg\phi}$ with the system to be verified
    * checking if this product is nonempty. (If the intersection is empty, the system does satisfy ϕ.)
  - Given a Büchi automaton B and RSM A, construct an RBA that accepts the intersection of the languages.

1.2.7 Complexity of LTL Model-Checking

- LTL model-checking of RSMs is EXPTIME-complete.
- LTL model-checking on finite-systems for $L(\diamond)$ (limited to arbitrary nesting of diamond and box; no next or until operators) is NP-complete.
- LTL model-checking on RSMs for $L(\diamond)$ is NP-complete.
- Model-checking of fragments that are PSPACE-complete on finite Kripke structures become EXPTIME-complete in RSMs. For all the remaining fragments this problem stys in the same complexity class on either RSMs or finite Kripke structures.

2 Attestor

2.1 What is Attestor?

- Verification tool for pointer programs in two steps:

1. Construction: Construct an abstract state space of the input program using graph grammars
    - Attestor requires a graph grammar to guide abstraction and concretization
    - Each state depicts links between heap objects and values of program variables using a graph representation
    - States include information about structural properties (reachability, heap shapes)
2. Verification: abstract state space is checked against a user-defined LTL specification
    - Provides counterexample in case of violations

2.2 Installation

- Get Attestor from `https://github.com/moves-rwth/attestor`
- To install maven dependencies, maven uses the settings file in `C:/Users/chau/.m2/settings.xml`.
  Need to disable (remove) it, so that maven does not use this file but default settings.
- To import the attestor project into Eclipse, import as Maven project.

2.3 Example Run

2.3.1 Setup

- Heaps are represented as graphs where objects are represented by nodes and pointers by edges between nodes.
- In a file .attestor or as command line options, specify classpath, class and method to be analysed
- Also requires graph grammar and initial heap configuration

2.3.2 Analysis

- Use Jimple to simplify Java Byte Code. Jimple statements represent the statements of each state.
- Each state is represented in an individual graph. It shows the current heap configuration in each step.

2.4 Recursive State Space Generation

`RecursiveStateSpaceGenerationPhase` ↔ Interprocecural Analysis

- Create instance of **interproceduralAnalysis**:
  - This class is responsible for computing the fixpoint of the interprocedural analysis in case there are recursive functions.
  - It keeps track of any **procedure calls** to recursive methods that have not yet been analysed.
  - stores the **dependencies** between **partialStateSpaces** and **procedureCalls** so that it can continue those stateSpaces whenever it has found new contracts for a procedureCall.
- **loadInitialStates**:
  - get initial heap configurations from input settings
  - create program states from heap configurations
- **loadMainMethod**:
  - get name of method to be analysed from input settings
  - get Method-object for method to be analyzed from scene
- **initializeMethodExecutors**:
  - InternalProcedureRegistry: interface for the **stateSpaceGeneration** (StateSpaceGeneratorFactory) to give relevant information to the **InterproceduralAnalysis** component. Interaction between semantics and analysis.
  - for each method, create a method executor (recursive or non-recursive) including contract collection and procedure registry for each method.
- **startPartialStateSpaceGeneration**

- `stateSpaceGeneratorFactory.create(mainMethod.getBody(), initialStates)`: create stateSpaceGenerator by setting program to analyze, initial states, and strategies.
- `stateSpaceGenerator.generate()`: starts state space generation
  - retrieve program state from state exploration strategy (DSF, starting with the initial program states)
  - for each state:
    - set the current state space as the containing state space for the program state
    - check if abort criteria is fulfilled, e.g., full memory, timeout
    - get semantics of program state
    - materialize if necessary (partially concretize an abstract heap configuration such that one step of the concrete semantics can be executed); materialized states are immediately added to the state space as successors of the given state
    - compute successor states of program state by executing a single step of the abstract program semantics on the given program state
    - **Special treatment for AssignInvoke and Invoke Statements**: differentiate between recursive and non-recursive method calls:
    - non-recursive: generate new state space for method (similar to main state space)
    - recursive: register procedure calls in procedure registry to be treated separately in InterproceduralAnalysis.run (only recursive procedure calls are processed here); procedure is **not** executed here yet
    - mark final states as final
    - handle successor states: add or merge states to state space (states might already exist if they are a continue state of a previously started state space)
- post processing
- add size of generated states to states counter

- **registerMainProcedureCalls**:

  - map procedure calls of the main method to the partial states spaces generated from initial program states in the interproceduralAnalysis

- **interproceduralAnalysis.run**
  - handle remaining *recursive* procedure calls and generate **new** state spaces/ contracts; registry registers new state space in interprocedural analysis
  - if all procedure calls are handled, handle partial state spaces that have not been finished yet
  - get unfinished state space and continue execution
  - if contracts have changed, notify according procedure call in order to continue its state space generation
  - add the partial state spaces to the remainingPartialStateSpaces queue

2.5 Model Checking

3 Algorithm for Hierarchical LTL Model Checking

- For a (recursive) input program $\mathcal{P}$ construct a respective Recursive State Machine $\mathcal{A}_\mathcal{P}$.
- Given a RSM $\mathcal{A}$ and an LTL formula φ check whether $\mathcal{A} \models \varphi$.
- If $\mathcal{A} \not\models \varphi$ return a counterexample and check whether it is spurious or not.

3.1 Translate input program $\mathcal{P}$ to a Recursive State Machine $\mathcal{A}_\mathcal{P}$.

(Jimple) Input program $\mathcal{P}$

- Input program $\mathcal{P}$ consists of a set of procedures (including an initial procedure (`main`)).
- Each procedure contains a set of statements.
- Statements can be

  - assignments
  - branching statements (`if/ else`, `skip`, `goto`)
  - invoke statements (procedure calls)
  - return statements (procedure returns)

- Each statement is referenced by a numerical program location.
- Global and local variables

Recursive State Machine (RSM) $\mathcal{A} = \langle A_1, ..., A_k \rangle$ over a finite alphabet Σ, where each component state machine $A_i = (N_i \cup B_i, Y_i, En_i, Ex_i, \delta_i)$ consists of

- a set N_i of nodes and a (disjoint) set B_i of boxes,
- a labeling $Y_i : B_i \mapsto \{1, \cdots, k\}$ that assigns to every box an index of one of the component state machines A_i, $1 \leq i \leq k$,
- a set of entry nodes $En_i \subseteq N_i$,
- a set of exit nodes $Ex_i \subseteq N_i$,
- a transition relation δ_i, where transitions are of the form (u, σ, v), where
  - the source u is either a node of N_i, or a pair (b, x), where b is a box in B_i and x is an exit node in Ex_j for $j = Y_i(b)$
  - the label σ is in Σ
  - the destination v is either a node in N_i or a pair (b, e), where b is a box in B_i and e is an entry node in En_j for $j = Y_i(b)$

Algorithm (Sketch)

- Given a (recursive) input program $\mathcal{P}$, compute an according RSM $\mathcal{A}_\mathcal{P}$ that represents the control flow in $\mathcal{P}$.
- Let k be the number of procedures in $\mathcal{P}$. Then $\mathcal{A}_\mathcal{P}$ has k component state machines, one for each procedure. Therefore, $\mathcal{A}_\mathcal{P} = \langle A_1, \cdots, A_k \rangle$.
- Each procedure i is represented by a respective component state machine $A_i = (N_i \cup B_i, Y_i, En_i, Ex_i, \delta_i)$, where
  - the set N_i of nodes composes of the program locations of procedure i. Each statement (except for procedure calls) is represented by a node $s \in N_i$. Index node s with respective program location of $\mathcal{P}$.

- For every distinct procedure j that is called by procedure i, we introduce a box $b \in B_i$ with $Y_i(b) = j$. If there are no procedure calls, $B_i = \emptyset$. Thus, procedure call statements are represented by boxes.
- The labeling Y_i is as described above.
- En_i composes of the entry point of procedure i.
- Ex_i composes of the return (exit) point(s) of procedure i.
- δ_i is determined by the control flow of $\mathcal{P}$.
  - internal transitions: transitions given by control flow graph that stay within procedure i
  - call transition: transitions from a node s to an entry node of a box b
  - return transition: transitions from exit node of a box b to the last node visited in the component that called b

3.2 Model check RSM $\mathcal{A}$ for LTL formula φ

Model checking RSM $\mathcal{A}$ happens on-the-fly: as soon as a counterexample is found, the process will be aborted. In order to save unnecessary computations, the state space of the program $\mathcal{P}$ (computation of heap configurations of each state of $\mathcal{A}$) will be computed on-the-fly as well.
Input: RSM $\mathcal{A}$, LTL formula φ
Output: `true` if $\mathcal{A} \models \varphi$, `false` and counterexample otherwise
Algorithm (Sketch)

- Let the component state machine A_1 of $\mathcal{A}$ refer to the `main`-method of the program $\mathcal{P}$. Let $en_1 \in En_1$ be the initial node/ entry node of A_1. Let en_1 be the current node.
- Get the current set of assertions Φ for formula φ.
- Get the respective program statement for the current node.
- Execute the program statement and compute the heap configuration HC for the current state,
  - according to the implemented methods in
    `InterproceduralAnalysis.run()`
    → `StateSpaceGenerator.generate()`
    → `stateSemanticsCommand.computeSuccessors(ProgramState)`
    which executes a single step of the abstract program semantics on the given program state and computes the set of successor states.

- After the computation of the heap configuration of the **current** state, model check its heap configuration HC for the current formula/ set of assertions using the tableau method.
  - need to change the tableau method implementation such that single states and a (sub)formula or a set of assertions can be taken as an input to the method call
  - should return model checking result, and next set of assertions
- If the tableau method returns a final value (`true` or `false`), the process is done.
- Otherwise, the next state according to the tableau method execution needs to be analysed. The next state to be checked is determined by the control flow of the program $\mathcal{P}$/ the transition relation of the RSM $\mathcal{A}$.
- If the next state is not a call or return state, the above procedure is repeated.
- If the next state enters procedure i, start a new state space for procedure i in order to generate contracts. Therefore, input last assertions of tableau method into the new model checking layer. Continue model checking procedure for internal nodes of procedure i.
- At the end (exit) of procedure i, return to calling position with updated assertions and model checking results.
- Compute contracts (pre-/ and post-conditions) after the whole state space of procedure i has been computed.
- Store model checking results (assertions that hold/ do not hold before/after model checking) together with contracts (as a kind of summary) in a table, so that the contracts can be reused in case procedure i is called with the same (sub)set of assertions at another program location.
- If the contract for procedure i has already been computed, but the assertion or formula φ has not been model checked and stored yet, start a new model checking subroutine for i and the stored state space. The state space of i does not need to be computed again. Add the model checking result to the table of model checking results of i.
- Continue this process until a counterexample has been found or the tableau method terminates.

3.2.1 Table of model checking results

In order to avoid multiple computation of the same model checking results for the same procedures, we introduce a table that stores the previously computed result together with pairs of pre- and post-conditions of a procedure in a table. Prior to model checking a procedure, we check

- Are pre-/ post-conditions available for the current procedure?
- Is there a matching set of pre-/ post-conditions to the current state?
- Has the set of assertions been model checked before for the current procedure?

If all above qeustions can be answered with 'yes', a table look up can check for the model checking results that must have been previously computed. Thus, the model checking process does not need to be repeated for the procedure and the set of assertions.

To-Do

- ☐ How can we structure the table such that a look up is as fast as possible?
- ☐ Is there a way to reuse or imply model checking results from set of assertions that do not coincide with the current set of assertions by 100%? e.g. expansion rules

3.3 Analyse counterexample for spuriosity

4 Implementation of Algorithm

4.1 RSMGenerationPhase

(2 weeks)

- Generate RSM from input program.
- Start with main method from input program. Main method is the method to be analyzed (not necessarily the main-method of the class).
- For every method, translate the statements to vertices, boxes, or component state machines.

- Edge relation is indirectly given by underlying control flow graph which returns the successor program state for each statement.
- Output is a RSM object for the input program.

Questions:

- Is it okay to use the control flow graph as the underlying edge relation or is it better to use a map to store edges explicitly in the RSM?
- Is it okay to use the classes Method, SemanticsCommand, ... ?
- Is the position of this phase good (after AbstractionPreprocessingPhase and before StateSpaceGeneration)?

4.2 ModelCheckingPhase

(2 weeks) Changed model checking method a little. Still using rulesSwitch, but handles Next-Formulae differently:

- A queue keeps track of the assertions to be expanded in the proof structure.
- The assertion object differentiates between formulae with a Next-operator and formulae without (two different lists).
- First, expand all formulae of a node without any Next-operator (ensured by insertion order of assertions into the assertion queue).
- If assertion has only Next-formulae left, handle Next-Formulae by applying expansion rule for Next-formulae and return new assertions for successor states of the current state. Here, we need a state switch.
- Successor states are currently taken from generated state space which is available during model checking phase.

4.3 On-the-Fly Model Checking

(2 weeks) Want to do model checking on on-the-fly generated state space. Thus, state space is not completely available during the model checking phase. Hence, successor states need to be generated on-the-fly. At the same time, want to be able to generate complete state space without model checking.

Possible approaches:

(A) Inject StateSpaceGenerator into model checking phase.
Problem: How can state space be generated without model checking?

(B) Trigger model checking from within the state space generator.
Problems:

- StateSpaceGenerator (StateExplorationStrategy) and Model Checker have own queues for keeping track of which states have been processed/ expanded. How can they be synchronized? Thought of computing all Next-fomulae for one state first before moving to next state, but this can get mixed up quite fast...
- Also need to keep track of parent assertions

4.3.1 Current solution

StateSpaceGenerator does not implement an own queue anymore, but asks proof structure for current state that needs to be generated and whose successors need to be computed.

On-the-fly model checking can be extended for non-recursive method calls by:

- Pass formulae set to new state space generator
- Generate new proof structure for method call 1
  - One state space for each method call vs. one big proof structure:
  - Procedures that have been seen before might not be model checked again: following states might be ignored
  - Big cycles crossing several state spaces can be reduced to cycles only within one state space since one state cannot repeat itself in another state space
  - need to map proof structure/ formulae to method calls for recursive methods (in interprocedural analysis)

Can also be adjusted for **recursive** method calls:

- Need to register "partial" proof structures in registry/ interprocedural analysis in order to map proof structures to method calls
- Model checking happens offline, not on-the-fly since state spaces could not be reused for recomputation

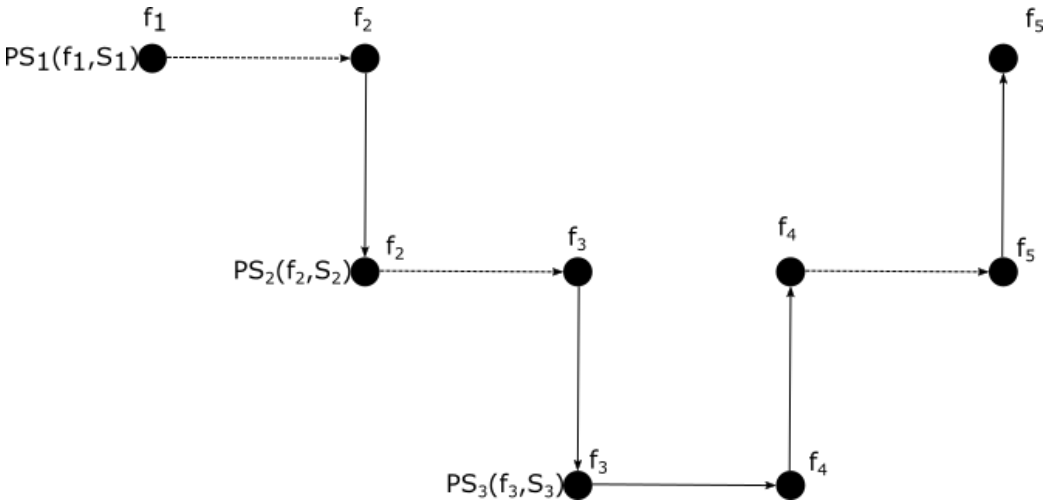

Figure 1: Model Checking Procedure.

On-the-fly model checking is basically based on the new method generateAndCheck() in StateSpaceGenerator class. Methods that use generate():

- ☐ CounterExampleGenerator
- ☑ InterproceduralAnalysis.run()
- ☑ InternalPartialStateSpace.continueExecution()
- ☑ InternalProcedureCall.execute()
- ☐ InternalContractGenerator
- ☐ AbstractMarkingGenerator

4.3.2 To-Do

- ☑ Set abort criteria
- ☑ Example: refSll should visit **all** states
- ☑ Example: MC_completeness_reverseWithInitialList has no final states: *Model checking terminates before reaching final states, thus they do not need to be generated*
- ☑ Generate recursive example
- ☑ Send back formulae resulting from model checking to upper state spaces

☑ Contract generation

- when are contracts used?
- failed Model Checking: abort, no contracts necessary
- successful Model Checking: complete state space needed, but maybe not generated if declared successful early (e.g. for Xf)
- **Generate complete state space if model checking was successful** and not complete state space generated yet:
  - mark states as explored in stateSpaceGenerator
  - continue generation with unexplored states –¿ generate()
  - generate post and preconditions
- return true when sub-proof structure is successful
- how to handle false proof structures?

☑ Store model checking results to avoid multiple checks of the same state space/ method → store results

☑ Adjust contract matching method to consider different input formulae for model checking

☐ Abort model checking as soon as a failure is found in a procedure state space (use boolean variable buildFullStructure to also allow for full model checking)

☐ Refactor state space generator

☐ Debug time issue of hierarchical model checking

☐ Reuse partial graphs for model checking (generate new states if needed) - map proof structure to the same (continuing) state space

☐ totalStatesCounter is at wrong position in generateAndCheck() method.. should only be called once for each state space

☐ Generate more examples for testing procedure model checking

4.3.3 Notes

- Procedure call examples: G {L(SLL)} fails if method with list object is called by another method, e.g. main because the main state space is not a list

- Constructor calls fail procedure model checking because they are started with initial empty heap; need to exclude them from procedure model checking and only use contracts on current heap

4.4 Hierarchical Model Checking (offline)

(2 weeks)

Possible approach:

- Create RSM from procedure state spaces
- Model check state spaces separately
- Get global failure trace

4.4.1 To-Do

☑ Implement recursive model checking in offline manner, but with hierarchical use of results: generate RSM out of procedure state spaces

4.5 Global To-Do's

☑ Implement mechanism to decide on which procedures are checked and which not: use option –mc-skip and –mc-mode to, "<init>" excludes all constructor calls

☐ Run Java Tests for proof structure

- ☑ TableauRulesSwitch2
- ☑ ProofStructure2 (SimpleProofStructure)
- ☐ HierarchicalProofStructure
- ☐ OntheflyProofStructure
- ☐ ModelCheckingContracts
- ☐ Mechanism to decide which procedures to check
- ☐ RSM

☐ Generate output report (diagram)

☐ Add statistics to model checking summary

☐ Clean up code

- [x] Write down implication relations for LTL formula (use for model checking contracts)
- [ ] Compare offline and on-the-fly model checking

4.6 Extensions

4.6.1 Statically imply model checking results from LTL-formulae

4.6.2 Pre-Model Checking of procedures/ contracts

(2 weeks) After procedure state space has been completely generated, model check state space for all sub-formulae of the initial LTL formula φ: $\mathcal{O}(3^n)$ sub-formulae where n is the number of sub-formulae in φ. – discard

4.6.3 Program generation from counterexample trace

(2 weeks) – discard