# Oracle-Guided Heap Invariant Synthesis

by Nishant Rajgopal Totla

## Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

**Committee:**

Sanjit Seshia
Research Advisor

Date

* * * * * *

Björn Hartmann
Second Reader

Date

**Abstract**

TODO

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Introduction

TODO

# Chapter 2

# Background

In this section, we define the formalize requirements for lazy interpolation-based model checking, based on [**?**]. This applies to the standard domain of model checking for data programs, and we'll extend it to heap-manipulating programs later.

We will use standard first-order logic (FOL) and the notation $\mathcal{L}(\Sigma)$ to denote the set of well-formed formulas (*wffs*) of FOL over a vocabulary $\Sigma$ of non-logical symbols. For a given formula or set of formulas $\phi$ we will use $\mathcal{L}(\phi)$ to denote *wffs* over the vocabulary of $\phi$.

For every non-logical symbol $s$, we presume the existence of a unique symbol $s'$ (that is, $s$ with one prime added). We think of $s$ with $n$ primes added as representing the value of $s$ at $n$ time units in the future. For any formula or term $\phi$, we will use the notation $\phi^{\langle n \rangle}$ to denote the addition of $n$ primes to every symbol in $\phi$ (meaning $\phi$ at $n$ time units in the future). For any set $\Sigma$ of symbols, let $\Sigma'$ denote $\{s' | s \in \Sigma\}$ and $\Sigma^{\langle n \rangle}$ denote $\{s^{\langle n \rangle} | s \in \Sigma\}$.

## 2.1 Modeling Programs

We use FOL formulas to characterize programs. To this end, let $S$, the state vocabulary, be a set of individual variables and uninterpreted $n$-ary functional and propositional constants. A *state formula*

is a formula in $\mathcal{L}(S)$ (which may also include various interpreted symbols, such as $=$ and $+$). A *transition formula* is a formula in $\mathcal{L}(S \cup S')$.

For our purposes, a *program* is a tuple $(\Lambda, \Delta, l_i, l_f)$, where $\Lambda$ is a finite set of program locations, $\Delta$ is a set of $actions$, $l_i \in \Lambda$ is the initial location and $l_f \in \Lambda$ is the error location. An $action$ is a triple $(l, T, m)$, where $l, m \in \Lambda$ are respectively the entry and exit locations of the action, and $G$ is a transition formula. A $path$ $\pi$ of a program is a sequence of transitions of the form $(l_0, T_0, l_1)(l_1, T_1, l_2) \cdots (l_{n-1}, T_{n-1}, l_n)$. The path is an *error path* when $l_0 = l_1$ and $l_n = l_f$. The unfolding $\mathcal{U}(\pi)$ of path $\pi$ is the sequence of formulas $T_0^{\langle 0 \rangle}, \cdots, T_{n-1}^{\langle n-1 \rangle}$, that is, the sequence of transition formulas $T_0, \cdots, T_{n-1}$, with each $T_i$ shifted $i$ time units into the future.

We will say that path $\pi$ is $feasible$ when $\bigwedge \mathcal{U}(\pi)$ is consistent. We can think of a model of $\bigwedge \mathcal{U}(\pi)$ as a concrete program execution, assigning a value ot every program variable at every time $0, \cdots, n-1$. A program is said to be $safe$ when every error path of the program is infeasible. An *inductive invariant* of a program is a map $I : \Lambda \to \mathcal{L}(S)$, such that $I(l_i) \equiv \mathsf{True}$ and for every action $(l, T, m) \in \Delta$, $I(l) \wedge T$ implies $I(m)'$. A *safety invariant* of a program is an inductive invariant such that $I(l_f) \equiv \mathsf{False}$. Existence of a safety invariant of a program implies that the program is safe.

To simplify the presentation of algorithms, we will assume that every location has at least one outgoing action. This can be made true without affecting program safety by adding self-loops.

## 2.2 Interpolants from Proofs

Given a pair of formulas $(A, B)$, such that $A \wedge B$ is inconsistent, an *interpolant* for $(A, B)$ is a formula $\hat{A}$ with the following properties:

- $A$ implies $\hat{A}$
- $\hat{A} \wedge B$ is unsatisfiable, and
- $\hat{A} \in \mathcal{L}(A) \cap \mathcal{L}(B)$

The Craig Interpolation lemma [**?**] states that an interpolant always exists for inconsistent formulas in FOL. To handle program paths, this idea can be generalized to sequences of formulas. That is, given a sequence of formulas $\Gamma = A_1, \cdots, A_n$, we say that $\hat{A}_0, \cdots, \hat{A}_n$ is an *interpolant* for $\Gamma$ when

- $\hat{A}_0 = \mathsf{True}$ and $\hat{A}_n = \mathsf{False}$ and,
- for all $1 \leq i \leq n$, $\hat{A_{i-1}} \wedge A_i$ implies $\hat{A}_i$ and
- for all $1 \leq i < n$, $\hat{A}_i \in (\mathcal{L}(A_1 \cdots A_i) \cap \mathcal{L}(A_{i+1} \cdots A_n))$

That is, the $i$-th element of the interpolant is a formula over the common vocabulary of the prefix $A_1 \cdots A_i$ and the suffix $A_{i+1} \cdots A_n$, and each interpolant implies the next, with $A_i$. If $\Gamma$ is quantifier-free, we can derive a quantifier-free interpolant for $\Gamma$ from a refutation of $\Gamma$, in certain interpreted theories [**?**].

## 2.3 Program Unwindings

We now give a definition of a program unwinding, and describe an algorithm to construct a complete unwinding using interpolants. For two vertices $v$ and $w$ of a tree, we will write $w \sqsubset v$ when $w$ is a proper ancestor of $v$.

**Definition 1** *An unwinding of a program $\mathcal{A} = (\Lambda, \Delta, l_i, l_f)$ is a quadruple $(V, E, M_v, M_e)$, where $(V, E)$ is a directed tree rooted at $\epsilon$, $M_v : V \to \Lambda$ is the vertex map, and $M_e : E \to \Delta$ is the edge map, such that:*

- $M_v(\epsilon) = l_i$
- *for every non-leaf vertex $v \in V$, for every action $(M_v(v), T, m) \in \Delta$, there exists an edge $(v, w) \in E$ such that $M_v(w) = m$ and $M_e(v, w) = T$*

**Definition 2** *A labeled unwinding of a program $\mathcal{A} = (\Lambda, \Delta, l_i, l_f)$ is a triple $(U, \psi, \rhd)$, where*

- *$U = (V, E, M_v, M_e)$ is an unwinding of $\mathcal{A}$*
- *$\psi : V \to \mathcal{L}(S)$ is called the vertex labeling, and*
- *$\rhd \subseteq V \times V$ is called the covering relation*

*A vertex $v \in V$ is said to be covered iff there exists $(w, x) \in \rhd$ such that $w \sqsubseteq v$. The unwinding is said to be safe iff, for all $v \in V$, $M_v(v) = l_f$ implies $\psi(v) \equiv$ False. It is complete iff every leaf $v \in V$ is covered.*

**Definition 3** *A labeled unwinding $(U, \psi, \rhd)$ of a program $\mathcal{A} = (\Lambda, \Delta, l_i, l_f)$, where $U = (V, E, M_v, M_e)$, is said to be well-labeled iff:*

- *$\psi(\epsilon) \equiv$ True, and*
- *for every edge $(v, w) \in E$, $\psi(v) \wedge M_e(v, w)$ implies $\psi(w)'$, and*
- *for all $(v, w) \in \rhd$, $\psi(v) \Rightarrow \psi(w)$, and $w$ is not covered*

*A vertex $v \in V$ is said to be covered iff there exists $(w, x) \in \rhd$ such that $w \sqsubseteq v$. The unwinding is said to be safe iff, for all $v \in V$, $M_v(v) = l_f$ implies $\psi(v) \equiv$ False. It is complete iff every leaf $v \in V$ is covered.*

Notice that, if a vertex is covered, all its descendants are also covered. Moreover, we do not allow a covered vertex to cover another vertex.

**Theorem 1** *If there exists a safe, complete, well-labeled unwinding of program $\mathcal{A}$, then $\mathcal{A}$ is safe.*

This is Theorem 1 from [**?**].

# Chapter 3

# Impact Algorithm

This section describes a semi-algorithm from [**?**], for building a complete, safe, well-labeled unwinding of a program. The algorithm terminates if the program is unsafe, but may not terminate if it is safe (which is expected, since program safety is undecidable). A non-deterministic procedure with three basic steps is outlined here. The three steps are

- EXPAND, which generates the successors of a leaf vertex (algorithm 1)
- REFINE, which refines the labels along a path, labeling an error vertex False (algorithm 2)
- COVER, which expands the covering relation (algorithm 3)

The interpoland in REFINE can be generated from a refutation of $\mathcal{U}(\pi)$ by the method of [**?**]. Each of the three steps preserves well-labeledness of the unwinding. To make the unwinding safe, we have to only apply REFINE to every error vertex. When none of the three steps can produce any change, the unwinding is both safe and complete, so we know that the original program is safe. To build a well-labeled unwinding, a strategy is required, for applying the three unwinding rules. The most difficult question is when to apply COVER. Covering one vertex can result in uncovering others. Thurs, applying COVER non-deterministically may not terminate.

## 3.1 Termination

TODO

Some discussion here about termination and and the UNWIND procedure

```
1 Procedure EXPAND(v ∈ V):
2     if v is an uncovered leaf then
3         foreach action (M_v(v), T, m) ∈ Δ do
4             add a new vertex w to V and a new edge (v, w) to E;
5             set M_v(w) ← m and ψ(w) ← True;
6             set M_e(v, w) ← T;
7         end
8     end
```

**Algorithm 1:** EXPAND: takes as input a vertex $v \in V$ and expands the control flow graph based on all actions available at that vertex.

**Procedure** `REFINE(`$v \in V$`)`*:*

- **if** $M_v(v) = l_f$ *and* $\psi(v) \not\equiv \mathsf{False}$ **then**
  - let $\pi = (v_0, T_0, v_1) \cdots (v_{n-1}, T_{n-1}, v_n)$ be the unique path from $\epsilon$ to $v$
  - **if** $\mathcal{U}(\pi)$ *has an interpolant* $\hat{A}_0, \cdots, \hat{A}_n$ **then**
    - **for** $i = 0 \cdots n$ **do**
      - let $\phi = \hat{A}_i^{\langle -i \rangle}$
      - **if** $\psi(v_i) \not\models \phi$ **then**
        - remove all pairs $(\cdot, v_i)$ from $\rhd$;
        - set $\psi(v_i) \leftarrow \psi(v_i) \wedge \phi$;
      - **end**
    - **end**
  - **else**
    - abort (program is unsafe)
  - **end**
- **end**

**Algorithm 2:** REFINE: takes as input a vertex $v \in V$ at an error location and tags the path from root to $v$ with invariants.

**Procedure** `COVER(`$v, w \in V$`)`*:*

- **if** $v$ *is uncovered and* $M_v(v) = M_v(w)$ *and* $v \not\models w$ **then**
  - **if** $\psi(v) \models \psi(w)$ **then**
    - add $(v, w)$ to $\rhd$;
    - delete all $(x, y) \in \rhd$, s.t. $v \sqsubseteq y$;
  - **end**
- **end**

**Algorithm 3:** COVER: takes as input vertices $v, w \in V$ and attempts to cover $v$ with $w$.

# Chapter 4

# Heap Patterms

To extend the Impact algorithm to heaps, we first define a framework for representing and reasoning about heaps.

In this section, we first define a standard language that performs low-level memory operations to update linked data structures (§4.1). We then review definitions of three-valued structures introduced in previous work [**?**], which we use to formulate patterns over program heaps (§4.4).

## 4.1 Language Definition

TODO : This subsection might end up being redundant later, if we choose to not use this LANG formalism, but it's here for now. We might later just choose to keep the heap and pattern definitions from this section.

In this section, we define the syntax (§4.2) and semantics (§4.3) of our subject language LANG.

$$\textsc{Lang} := (\texttt{Locs} : \texttt{Instrs})^* \tag{4.1}$$

$$\texttt{Instrs} := \mathsf{instr}_P \mid \mathsf{instr}_H \tag{4.2}$$

$$\mathsf{instr}_P := \texttt{Vars}_P := \texttt{Vars}_P\ \texttt{Ops}_P\ \texttt{Vars}_P \tag{4.3}$$

$$\mid \texttt{Vars}_P := (\texttt{Vars}_H = \texttt{Vars}_H) \tag{4.4}$$

$$\mid \texttt{br}\ \texttt{Vars}_P, \texttt{Locs}, \texttt{Locs} \tag{4.5}$$

$$\mathsf{instr}_H := \texttt{Vars}_H := \texttt{alloc}() \tag{4.6}$$

$$\mid \texttt{Vars}_H := \texttt{Vars}_H \tag{4.7}$$

$$\mid \texttt{Vars}_H := \texttt{Vars}_H \rightarrow \texttt{Fields} \tag{4.8}$$

$$\mid \texttt{Vars}_H \rightarrow \texttt{Fields} := \texttt{Vars}_H \tag{4.9}$$

Figure 4.1: Syntax of heap-updating programs, LANG. The spaces of control locations, predicate variables, heap variables, and fields are denoted $\texttt{Locs}$, $\texttt{Vars}_P$, $\texttt{Vars}_H$, and $\texttt{Fields}$, respectively.

## 4.2 Syntax

A LANG program is a sequence of instructions that operate on a fixed set of predicate variables and pointers to heap objects. The syntax of LANG is given in Figure 4.1 for fixed finite sets of control locations $\texttt{Locs}$, predicate variables $\texttt{Vars}_P$, heap variables $\texttt{Vars}_H$, and heap fields $\texttt{Fields}$. A program is a sequence of instruction, each labeled with a control location (Equation 4.1). An instruction either updates the program's predicate variables or heap variables (Equation 4.2). An instruction that updates predicate variable either stores in a predicate variable the result of a Boolean operation (Equation 4.3), an equality test on heap cells (Equation 4.4), or branches control based on the value in a predicate variable (Equation 4.5). An instruction that updates heap variables either allocates a new heap cell (Equation 4.6), copies the heap cell from one pointer variable to another (Equation 4.7), loads a heap cell into a pointer variable (Equation 4.8), or stores a heap cell as a child of a cell in a pointer variable (Equation 4.9).

$$\text{ALLOC}\frac{\begin{array}{cc} n \notin N & N' = n \cup \{N\} \\ V' = V[\mathtt{h} \mapsto n] & F' = F[\{(n,f) \mapsto \mathsf{nil}\}_{f \in \mathtt{Fields}}]\end{array}}{\langle (C,V,F), \mathtt{h} := \mathtt{alloc}() \rangle \to (N',V',F')} \qquad \text{COPY}\frac{V' = V[\mathtt{g} \mapsto V(\mathtt{h})]}{\langle (C,V,F), \mathtt{g} := \mathtt{h} \rangle \to (C,V',F)}$$

$$\text{LOAD}\frac{V' = V[\mathtt{p} \mapsto F(V(\mathtt{q}),\mathtt{f})]}{\langle (C,V,F), \mathtt{p} := \mathtt{q}{\to}\mathtt{f} \rangle \to (C,V',F)} \qquad \text{STORE}\frac{F' = F[(\mathtt{p},\mathtt{f}) \mapsto V(\mathtt{q})]}{\langle (C,V,F), \mathtt{p}{\to}\mathtt{f} := \mathtt{q} \rangle \to (C,V,F')}$$

Figure 4.2: Inference rules that define $\to_H$, the transition relation over heaps and heap updates.

## 4.3 Semantics

A LANG program updates a state, which consists of a control location, evaluation of predicate variables, and a heap, which is a graph with labeled edges.

Also refer to Figure 4.2.

## 4.4 Heap pattern Language

**Definition 4** *A* LANG heap *is a triple* $(C, V_H, F)$*, where*

1. *For the countable universe of heap cells, denoted* $\mathcal{C}$*,* $C \subseteq \mathcal{C}$ *is a finite set of cells that contains a distinguished cell* $\mathsf{nil} \in \mathcal{C}$*.*
2. $V_H : \mathtt{Vars}_H \to C$ *maps each heap variable to the cell that it stores. We denote the space of evaluations of heap variables as* $\mathsf{Evals}_H = \mathtt{Vars}_H \to \mathcal{C}$*.*
3. $F : C \times \mathtt{Fields} \to C$ *maps each cell* $c \in C$ *and field* $\mathtt{f} \in \mathtt{Fields}$ *to the child of* $c$ *at* $\mathtt{f}$*. We denote the space of field maps as* $\mathsf{FieldMaps} = \mathcal{C} \times \mathtt{Fields} \to \mathcal{C}$*.*
4. $\mathsf{PredLbl} : C \to 2^{\mathtt{Vars}_P}$ *maps each cell* $c \in C$ *to an assignment of Boolean values to* $\mathtt{Vars}_P$*. We denote the space of predicate labeling functions as* $\mathsf{PredLblFn} = \mathcal{C} \to 2^{|\mathtt{Vars}_P|}$*.*

*We denote the space of heaps as* $\mathsf{Heaps} = \mathcal{P}(\mathcal{C}) \times \mathsf{Evals}_H \times \mathsf{FieldMaps} \times \mathsf{PredLblFn}$*, and the space of characteristic functions of languages of heaps as* $\mathsf{Chars} = \mathsf{Heaps} \to \mathbb{B}$*.*

A heap pattern is a labeled graph whose nodes and edges model the cells and fields of potentially-many heaps. The nodes and edges of a pattern are annotated with three-valued truth values that represent if all cells modeled by a node either definitely are, definitely are not, or may be (1) stored in a given heap variable or (2) connected by a heap field. The heap patterns are based on three-valued structures introduced in previous work on shape analysis [**?**].

**Definition 5** *Let the domain of* three-valued truth values *be* $\mathbb{B}_3 = \{\mathsf{True}, \mathsf{False}, \mathsf{Maybe}\}$. *A* heap pattern *is a labeled graph* $(N, \mathtt{V}, \mathtt{P}, E, \sigma)$*, where:*

1. *For the countable universe of nodes* $\mathcal{N}$*,* $N \subseteq \mathcal{N}$ *is a finite set of nodes.*
2. $\mathtt{V} : N \times \mathtt{Vars}_H \to \mathbb{B}_3$ *is a* heap-variable labeling. *We denote the space of heap-variable labelings as* $\mathsf{VarLbls} = \mathcal{N} \times \mathtt{Vars}_H \to \mathbb{B}_3$.
3. $\mathtt{P} : N \times \mathtt{Vars}_P \to \mathbb{B}_3$ *is a* predicate-variable labeling. *We denote the space of predicate-variable labelings as* $\mathsf{PredLbls} = \mathcal{N} \times \mathtt{Vars}_P \to \mathbb{B}_3$.
4. $E : N \times \mathtt{Fields} \times N \to \mathbb{B}_3$ *is a set of labeled* edges. *We denote the space of labeled edges as* $\mathsf{LblEdges} = \mathcal{N} \times \mathtt{Fields} \times \mathcal{N}$.
5. $\sigma : N \to \mathbb{B}$ *is a summary-labeling for nodes. A node is a summary node if it can be used to represent more than one concrete node.*

*We denote the class of all heap patterns as* $\mathsf{Pats} = \mathcal{P}(\mathcal{N}) \times \mathsf{VarLbls} \times \mathsf{PredLbls} \times \mathsf{LblEdges}$.

Program in a language with low-level memory updates, such as `alt_list` (**??**), can be modeled as LANG programs if a verifier is provided with a bounded set of "relevant" predicates $R$ over heap cells. In such a case, the verifier can simply model the predicates in $R$ as heap fields. To simplify the presentation of our analysis, we assume that a separate program analysis has inferred such a set of relevant predicates, and that such predicates are already modeled as LANG fields.

**Example 1** *Pattern* $P$ *(**??**, **??**) is a heap pattern that contains two summary nodes* $s_0$ *and* $s_1$*, and one non-summary node* $n$*. For summary node* $s_0$*, predicate* `data!=d` *maps to* True *and all other*

*predicates map to* False *For summary node* $s_1$*, predicate* `data=d` *maps to* True*, predicate* `l` *maps to* Maybe*, and all other predicates map to* False*. For concrete node* $n$*, predicate* `EqNil` *maps to* True *and all other predicates map to* False*.*

*The edges* $(s_0, s_1)$*,* $(s_0, n)$*,* $(s_1, s_0)$*, and* $(s_1, n)$ *map on field* `next` *to* Maybe*. All other edges on all other fields map to* False*.*

Each heap $h$ defines a heap pattern that represents exactly $h$.

**Definition 6** *For each heap* $h = (C, V_H, F)$*, the* concrete pattern *of* $h$ *is* $G_h = (N_h, \mathtt{V}_h, E_h)$*, where:*

1. *Each node in* $G_h$ *is a cell in* $C$*. I.e.,* $N_h = C$*.*
2. *Each heap-variable binding in* $V_H$ *defines a node labeling in* $\mathtt{V}_h$*. I.e., for each node* $c \in C$ *and heap variable* $\mathtt{p} \in \mathtt{Vars}_H$*, if* $V_H(\mathtt{p}) = c$*, then* $\mathtt{V}_h(c, \mathtt{p}) =$ True*, and otherwise,* $\mathtt{V}_h(c, \mathtt{p}) =$ False*.*
3. *Each field binding in* $F$ *defines an edge in* $E_h$*. I.e., for all cells* $c, c' \in C$ *and each field* $\mathtt{f} \in \mathtt{Fields}$*, if* $c' = F(c, \mathtt{f})$*, then* $E_h(c, \mathtt{f}, c') =$ True*, and otherwise,* $E_h(c, \mathtt{f}, c') =$ False*.*

**Example 2** **??**, **??** *contains concrete patterns for six heaps: three alternating heaps and three non-alternating heaps.*

**Definition 7** *Given two heap patterns* $P_1$ *and* $P_2$*, we say that* $P_1 \models P_2$ *(*$P_1$ *entails* $P_2$*) if every heap represented by* $P_1$ *is also represented by* $P_2$*.*

The algorithm for checking entailment is described in §4.5

The entailment relation over heap patterns formulates under both (1) under what conditions a heap pattern describes a heap and (2) under what conditions all of the heaps described by one heap pattern are described by another pattern.

**Definition 8** *Let the* information-precision ordering $\sqsubseteq_3 \subseteq \mathbb{B}_3 \times \mathbb{B}_3$ *be such that* $\mathsf{True} \sqsubseteq_3 \mathsf{Maybe}$ *and* $\mathsf{False} \sqsubseteq_3 \mathsf{Maybe}$.

*For all heap patterns* $P, P' \in \mathsf{Pats}$ *with* $P = (N, \mathtt{V}, E)$ *and* $P' = (N', \mathtt{V}', E')$, $P$ entails $P'$, *denoted* $P \preceq P'$, *if there is a map* $h : N \to N'$ *such that:*

- $h$ *embeds heap-variable assignments. I.e., for each pattern node* $n \in N$, $\mathtt{V}(n) \sqsubseteq_3 \mathtt{V}'(h(n))$.
- $h$ *embeds field labelings. I.e., for all pattern nodes* $n_0, n_1 \in N$ *and fields* $\mathtt{f} \in \mathtt{Fields}$, $E(n_0, \mathtt{f}, n_1) \sqsubseteq_3 E'(h(n_0), \mathtt{f}, h(n_1))$.

For each heap $h \in \mathsf{Heaps}$ and pattern $P \in \mathsf{Pats}$, if the concrete pattern (Defn. 6) of $h$ entails $P$, then $h$ is *modeled* by $P$, which we alternatively denote as $h \preceq P$. For any two heap patterns $P_0, P_1 \in \mathsf{Pats}$ and heap $h \in \mathsf{Heaps}$, if $P_0 \models P_1$ and $h \preceq P_0$, then $h \preceq P_1$.

**Definition 9** *In* **??**, **??**, *each of the patterns* $A_0$, $A_1$, *and* $A_2$ *for an alternating heap entails the pattern* $P$. *A matching from* $A_1$ *to* $P$ *is depicted as dotted arrows from the nodes of* $A_1$ *to the nodes of* $P$.

## 4.5 Pattern Entailment Algorithm

TODO: The entailment algorithm is based on simulation checks, and is very similar. It is described here.

## 4.6 Modeling Heap Programs

TODO: This section ties up the description of heap patterns back to the program modeling discussion in §2.1. In short, the formalism differs in the following way

**Definition 10** *A labeled unwinding of a program* $\mathcal{A} = (\Lambda, \Delta, l_i, l_f)$ *is a triple* $(U, \psi, \triangleright)$, *where*

- $U = (V, E, M_v, M_e)$ *is an unwinding of* $\mathcal{A}$
- $\psi : V \to \mathsf{Pats}$ *is called the vertex labeling, and*
- $\rhd \subseteq V \times V$ *is called the covering relation*

*Note that the major difference is that now vertices* $v \in V$ *contain a heap pattern, instead of an FOL formula.*

# Chapter 5

# Heat Impact Algorithm

Building on top of the framework defined in §2, §3, and §4, we can now modify the lmpact algorithm to work for heap-manipulating programs. In this section, we first define the three steps of lmpact , that is EXPAND, COVER, and REFINE. Then we describe an invariant-learning procedure that retrieves patterns from an Oracle, thereby completing the description of the algorithm.

```
1 Procedure EXPAND(v ∈ V):
2     if v is an uncovered leaf then
3         foreach action (M_v(v), T, m) ∈ Δ do
4             add a new vertex w to V and a new edge (v, w) to E;
5             set M_v(w) ← m and ψ(w) ← 1_D;
6             set M_e(v, w) ← T;
7         end
8     end
```

**Algorithm 4:** EXPAND: takes as input a vertex $v \in V$ and expands the control flow graph based on all actions available at that vertex.

## 5.1 Postcondition Transforms for Heap Operations

TODO: This section needs to be completed.

```
1  Procedure REFINE(v ∈ V):
2      if M_v(v) = l_f and ψ(v) ≢ (False, 0_D) then
3          let π = (v_0, T_0, v_1) ··· (v_{n−1}, T_{n−1}, v_n) be the unique path from ε to v
4          let Â_0, ··· , Â_n = INVLEARNER (U(π))
5          if Â_0, ··· , Â_n is a valid interpolant then
6              for i = 0 ··· n do
7                  let φ = Â_i^{⟨−i⟩}
8                  if ψ(v_i) ⊭ φ then
9                      remove all pairs (·, v_i) from ▷;
10                     set ψ(v_i) ← ψ(v_i) ∧ φ;
11                 end
12             end
13         else
14             abort (program is unsafe)
15         end
16     end
```

**Algorithm 5:** REFINE: takes as input a vertex $v \in V$ at an error location and tags the path from root to $v$ with invariants.

**Definition 11** *We define the operator* Post*, which computes the strongest postcondition for a given heap pattern, and action. That is* $\mathsf{Post} : \mathsf{Pats} \times \mathcal{T} \to \mathsf{Pats}$*, where* $\mathcal{T}$ *is the set of all actions.*

*The* Post$*$ *operator can be defined as a repeated application of* Post *along a given path. More formally, it is* $\mathsf{Post}* : \mathsf{Pats} \times \mathcal{P} \to \mathsf{Pats}$*, where* $\mathcal{P}$ *is a path in the unwinding.*

The formal rules for computing Post for each individual action are presented in **??**. (TODO)

## 5.2 Learning Invariants from Positive and Negative Examples

We note in algorithm 5 the procedure INVLEARNER is used, which is the core component of making Impact work for heap-manipulating programs. In this section, we describe this algorithm.

**Procedure** COVER($v, w \in V$):
  **if** $v$ *is uncovered and* $M_v(v) = M_v(w)$ *and* $v \not\models w$ **then**
    **if** $\psi(v) \vDash \psi(w)$ **then**
      add $(v, w)$ to $\rhd$;
      delete all $(x, y) \in \rhd$, s.t. $v \sqsubseteq y$;
    **end**
  **end**

**Algorithm 6:** COVER: takes as input vertices $v, w \in V$ and attempts to cover $v$ with $w$.

We defined the notion of interpolants in §2.2, and the same applies to INVLEARNER, which is described in algorithm 7.

**Procedure** INVLEARNER($\mathcal{U}(\pi)$):
  Let $\pi = (l_0, T_0, l_1)(l_1, T_1, l_2) \cdots (l_{n-1}, T_{n-1}, l_n)$
  Set $\hat{A}_i = 1_D, 0 \leq i < n, \hat{A}_n = 0_D$
  Set $H_i^+ = \{\}, 0 \leq i \leq n$
  Set $H_i^- = \{\}, 0 \leq i \leq n$
  **while** $\neg$ISINTERPOLANT $(\hat{A}, \mathcal{U}(\pi))$ **do**
    pick $i \in \{1, 2, \cdots, n-1\}$ $\hat{A}_i =$ NEWCANDIDATE$(l_i, \hat{A}, H^+, H^-, \psi, \mathcal{U}(\pi))$
  **end**
  **return** $\hat{A}_0, \hat{A}_1, \cdots, \hat{A}_n$

**Algorithm 7:** INVLEARNER: takes as input an unfolding $\mathcal{U}(\pi)$ of path $\pi$ and attempts to find an invariant for it.

While INVLEARNER is the higher level procedure to find an interpolant, it uses a sub-procedure called NEWCANDIDATE as a feedback loop with the Oracle, to accept candidate heap patterns that can be used to construct an interpolant.

**Procedure** ISINTERPOLANT($\hat{A}, \mathcal{U}(\pi)$):
  **if** $\hat{A}_0, \hat{A}_1, \cdots, \hat{A}_n$ *is an interpolant for* $\mathcal{U}(\pi)$ **then**
    **return** True
  **end**
  **return** False

**Algorithm 8:** ISINTERPOLANT: takes as input candidates $\hat{A}$ and unfolding $\mathcal{U}(\pi)$ of path $\pi$, and checks if $\hat{A}$ represents an interpolant for the unfolding.

**Procedure** NEWCANDIDATE($l_i, \hat{A}, H^+, H^-, \psi, \mathcal{U}(\pi)$):
  Let $\pi = (l_0, T_0, l_1)(l_1, T_1, l_2) \cdots (l_{n-1}, T_{n-1}, l_n)$
  Set $S = \mathsf{Post}^*(1_D, \pi_{0,i})$
  Set $C = 1_D$ **while** True **do**
    $\mathcal{C} = \mathcal{O}(H_i^+, H_i^-)$
    **if** $S \not\models C$ **then**
      $H_i^+ = \{h\} \cup H_i^+$ where $h \in S, h \notin C$
      continue
    **end**
    **if** $TODO$ **then**
      // TODO case add negative example $H_i^- = \{h\} \cup H_i^-$ where $h \cdots$
      continue
    **end**
    break
  **end**
  **return** $C$

**Algorithm 9:** NEWCANDIDATE: takes as input a program location $l_i$, current set of candidates $\hat{A}$, sets of positive and negative examples for each location ($H^+, H^-$ respectively), map $\psi$, and unfolding $\mathcal{U}(\pi)$ of path $\pi$, and interacts with the Oracle $\mathcal{O}$ to find a new candidate for $l_i$.

# Chapter 6

# Example

In this section, we demonstrate how the Impact algorithm for heap-manipulating programs would work with an example program.

# Bibliography