Oracle-Guided Heap Interpolant Synthesis

Nishant Totla

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-158
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-158.html

October 19, 2016

Oracle-Guided Heap Interpolant 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

We consider the problem of verifying safety properties of heap-manipulating programs. A central challenge in such verification is to infer auxiliary invariants on the heap that enable one to prove the desired goal property. Inferring such invariants is tricky for current software verifiers due to the limitations of deductive proof engines for existing heap logics to automatically extract them during proof search. In this thesis, we propose an alternative oracle-guided approach for heap interpolant synthesis. Our approach reduces the verification problem to one of inductive synthesis, where the safety verification problem is reduced to one of synthesizing a heap separator pattern from positive and negative examples of heaps. This reduction makes our approach modular by offloading the invariant synthesis step to an external oracle. This oracle can be implemented in various ways, with varying degrees of expressiveness and automation. We demonstrate one example of such an external oracle, a human who can interact with the prover by looking at positive and negative examples of concrete heaps, and propose generalized heap patterns.

Contents

List of Figures

Acknowledgements

There are several people I'd like to thank for making this thesis possible. First and foremost, my advisor Prof. Sanjit Seshia, who encouraged me to pursue my own research ideas, and was constantly supportive and patient with the many setbacks in my eventually doomed graduate career. I'd also like to thank Prof. Bill Harris, Prof. Somesh Jha, and Drew Davidson, who have had a major role to play in defining and concretizing the ideas presented in this thesis. It is only due to our sustained collaboration that this work managed to see the light of day. Also thanks to Rohit Sinha, who was heavily involved in the conception and inital work on these ideas. I would also like to thank Prof. Björn Hartmann, whose valuable comments made this thesis much more robust and well-rounded. I'd also like to thank him and Sanjit for being considerate about the difficult circumstances surrounding the conclusion of this work.

Before I thank others, I must describe how working on this thesis (and graduate school) affected me personally. I will forever remember this time as one of the darkest periods of my life. It was partly due to severe personal difficulties that led to chronic and often self-destructive depression. It was a constant hindrance towards my ability to make progress with research. It made a notable dent in relationships with friends and family, but I'd like to thank those who stood by me. First of all, I'd like to thank my girlfriend Vasuki, who has had to bear the most immediate impact of this dark period. Despite her own issues, she was often the only one who truly listened to me, and made a conscious effort to understand my problems and help me in concrete ways. A lot of the motivation to finish this work and move on in life wouldn't have been possible without her. I'd also like to thank my parents, who have always supported me in everything I chose to do, including my successes and failures, and positively encouraged me to never underestimate my abilities. I must thank my other friends in Berkeley – Avinash, Vaishnavi, Debanjan, Sharanya, and Vikram who provided me with the sense of community that invariably played a large role in surviving this painful phase. Avinash in particular was a constantly calming presence in the crunch period when I missed the final submission deadline and was facing severe consequences. I'd also like to thank a few of my other friends – Anasuya, Abhay, Pradeep, Pramod, Vishal, Namit, and Ravi, who despite

being physically further away, understood my pain and frustration. My conversations with Anasuya in the worst of times have been an invaluable source of stability.

Finally, I'd like to thank the extremely helpful people at the Disabled Students Program (DSP) in Berkeley and my therapist, without whose help I'd have had to pay thousands of dollars to submit the thesis.

Chapter 1

Introduction

In the context of hardware and software systems, formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics. Formal verification can be helpful in proving the correctness of systems such as: cryptographic protocols, combinational circuits, digital circuits with internal memory, and software expressed as source code. Verification uses a wide variety of techniques, in particular logic calculi, formal languages, automata theory, and program semantics, but also type systems and algebraic data types to problems in software and hardware specification and verification.

The verification of these systems is done by providing a formal proof on an abstract mathematical model of the system, the correspondence between the mathematical model and the nature of the system being otherwise known by construction. Examples of mathematical objects often used to model systems are: finite state machines, labeled transition systems, Petri nets, vector addition systems, timed automata, hybrid automata, process algebra, formal semantics of programming languages such as operational semantics, denotational semantics, axiomatic semantics and Hoare logic.

1.1 Software Verification

The goal of software verification is to assure that software fully satisfies all the expected requirements. Software verification has been successful for improving the quality of computer programs. Several fundamental concepts were invented in the last decade which made it possible to scale the technology from tiny examples to real programs. Predicate abstraction [BMMR01] with counterexample guided abstraction refinement (CEGAR) [CGJ+03] and lazy abstraction [HJMS02] is one such set of techniques. Lazy abstraction with Interpolants [McM06] is another approach. Several of these approaches are flexible and can be specialized or extended for sub-classes of programs or specifications.

1.2 Verifying Heap-manipulating Programs

A heap-manipulating program is one that updates heap memory using low-level memory operations (such as allocating or deallocating memory, or modifying pointers). Dealing with programs with pointers and dynamic linked data structures is among the most challenging tasks of formal analysis and verification due to a need to deal with infinite sets of reachable program configurations having the form of complex graphs. This task becomes even more complicated when considering low-level memory operations such as pointer arithmetic, safe usage of pointers with invalid targets, block operations with memory, reinterpretation of the memory contents, or address alignment. Despite the rapid progress in the area of formal program analysis and verification, fully automated approaches capable of efficiently handling sufficiently general classes of dynamic linked data structures in the form used in low-level code are still missing.

1.3 Oracle-guided Heap Interpolant Synthesis

Traditional formal methods do not scale to the size of software found in modern computer systems. This is further exacerbated in the case of heap-manipulating programs. Verification also currently requires highly specialized engineers with deep knowledge of software technology and mathemat-

ical theorem-proving techniques. These constraints make current formal verification techniques expensive and time-consuming.

The formal verification community has relentlessly pushed for automation, and it is indeed the right strategy for many problems. But for many other problems, human insight and involvement remain crucial. For instance, the steps of writing a specification, typically in the form of properties (assertions), creating an environment model, typically as input constraints or a state machine currently both require significant human intervention. Even the task of running the verifier, such as a model checker, which is usually thought of as a "push-button" step might require human insight such as hints to the verifier in the form of suitable abstraction techniques or inductive invariants. If the verifier returns a counterexample trace, then analysing it and finally fixing the design to debug the program also needs human input. Despite the focus on automating the verification process, we continue to need human insight in a variety of tasks, including writing specifications, creating models, guiding the verification engine, debugging and error localization, and repair.

It might be a while before we have verification systems that don't require any human expertise, but an alternative approach to consider is to change the way humans provide insight to the verifier. Today, mostly domain experts interact with verification tools, but such experts are few and expensive. Besides, even experts might sometimes struggle to make progress at different stages of the algorithm. We believe that the experts and automated tools can be assisted in the verification process by a crowd of non-expert humans performing relatively simpler tasks. Each task might involve pattern recognition or other cognitive operations that humans are typically good at. The main challenges are to recognise steps in the verification process where human insight is useful, find ways to transform these steps into tasks that non-expert humans can perform, and finally combine the results of these tasks to use in the verification process.

Along these lines, our work describes ProveIt, an Oracle-guided verification algorithm for proving the safety of heap-manipulating programs. We present an extension of the interpolant-based verification algorithm in [McM06] to work for heaps, along with an interface that collects insight from an external Oracle. In our case, the Oracle is a human user (although it could be any other Oracle that can interact with the verifier). The user plays a graphical game that abstracts away details of the verification algorithm, allowing non-experts to play. Essentially, our system is a verification tool for heap-manipulating programs where crucial input is collected from human users.

1.4 Outline

The rest of this thesis is organized as follows. §2 gives a high-level overview of our approach with an example, laying an intuitive foundation for the formal description that follows. §3 presents an extensive summary of the framework for finding inductive invariants from interpolants, and the Impact algorithm from [McM06]. §4 describes our formalism for representing heaps and heap patterns, and modeling heap-manipulating programs for verification. §5 gives a detailed description of our main algorithm PROVEIT, as an extension of the Impact algorithm. Our algorithm requires an Oracle, which we have chosen to be a human user, and §6 presents our user interface with some analysis. §7 discusses related work, and concludes.

Collaboration

The work in this thesis is part of the collaboration with Somesh Jha[1], William (Bill) Harris[2], and Drew Davidson[3]. Somesh and Bill were immensely insightful in helping narrow down the problem domain. The heap pattern formalism that forms a major part of the formal background for this thesis was conceived over the course of several regular weekly meetings with Bill, Somesh, and Sanjit. This involved understanding and analyzing several existing approaches, and narrowing down on the properties we cared about for an oracle-guided framework. They were also instrumental in advancing theoretical analysis of heap patterns (this is still work in progress, and didn't make it into the thesis).

Drew and I worked together implementing our ideas, initially building on top of Predator [DPV13], and later in CPACHECKER [BK09]. Later on, I worked on finishing this implementation, and also independently built the user interface to human oracles §6.

This work would not have been possible without our regular weekly discussions, constant guidance from Bill, Somesh, and Sanjit, and the shared effort with Drew on implementing these ideas.

[1] University of Wisconsin, Madison
[2] Georgia Institute of Technology
[3] University of Wisconsin, Madison

Chapter 2

Overview

We give an informal overview of the approach that we propose in this thesis. In §2.1, we introduce an example program `alt_list`, based on a benchmark in the SV-COMP [sv-15] benchmark suite, that updates its heap using low-level memory operations. In §2.2, we review a class of heap patterns that represent sets of heaps of unbounded size. In §2.3, we present a class of proof structures that use the heap patterns introduced in §2.2 to represent program invariants in order to prove that low-level heap programs, such as `alt_list`, satisfy their desired assertions. In §2.4, we define a class of learning games of program heaps and heap patterns. In §2.5, we describe how a program verifier can reduce the problem of constructing a valid proof of program safety to winning a set of the learning games described in §2.4.

2.1 An example low-level heap program

A low-level heap program is one that updates heap memory using low-level memory operations (such as allocating or deallocating memory, or modifying pointers). Figure 2.1 contains the source code for a program `alt_list`, written in a C-like, low-level language. For the states of `alt_list`, let a list be *alternating* if (1) the data field of the head of the list is equal to the value stored in Boolean variable `d` and (2) the data fields in successive cells of the list store alternating Boolean values. `alt_list` does two things:

```
1  void alt_list() {
2    bool d = TRUE;
3    List l = cons(d, NIL);
4    // LOOP-CONS: build a list with alternating Boolean values.
5    while (non_det()) {
6      d = !d;
7      l = cons(d, l);
8    }
9    // LOOP-CHK: check that the Boolean values alternate.
10   while (l != NIL) {
11     assert(l->data == d);
12     d = !d;
13     l = l->next;
14   }
15   return;
16 }
```

Figure 2.1: `alt_list`: a simplified version of an SV-COMP benchmark program that (1) constructs a list with cells that store alternating Boolean values and (2) checks that the Boolean values in successive cells alternate.

1. `LOOP-CONS`: iteratively constructs an alternating list of non-deterministic length, and then
2. `LOOP-CHK`: checks that the constructed list is indeed alternating

`alt_list` initializes the list stored in `l` to consist of a single cell whose value is equal to `d` and whose successor cell is `NIL` (the function `cons` takes as input a Boolean value d and a list cell l and returns a new list cell whose data field stores d and whose `next` field stores l). `alt_list` then non-deterministically chooses whether to execute `LOOP-CONS`, prepends a new cell to the list stored in `l` (line 5). If `alt_list` chooses to execute `LOOP-CONS`, then it negates the value stored in Boolean variable `data` (line 6), and constructs a new list cell whose next cell is the cell stored in `l` and whose data value is stored in `data` (line 7).

After exiting `LOOP-CONS`, `alt_list` executes `LOOP-CHK` to iteratively check that the values in successive cells of `l` store alternating Boolean values. In each iteration, `alt_list` checks if `l` stores `NIL` (line 10). If not, then `alt_list` checks that the data value of `l` is equal to `data` (line 11), and if so, inverts the value stored in `data` (line 12) and updates `l` to store its successor (line 13). Otherwise, if `l` stores `NIL`, then `alt_list` returns successfully (line 15).

The problem that we address in this thesis is to determine if a given low-level heap program, such as `alt_list`, always satisfies each of its assertions, such as the assertion on line 11, which checks that the list stored in `l` is alternating.

2.2 Representing sets of heaps with graph patterns

In this work, we propose a verifier, PROVEIT, that represents sets of heaps as graph *patterns*, which are directly analogous to three-valued structures introduced in previous work on shape analysis [SRW02]. In this chapter, we review graph patterns as they have been presented in previous work, in particular how they are used to represent sets of program heaps.

In our approach, each program heap is modeled as a labeled graph (we call it a heap graph), in which each node models a heap cell, and is labeled with facts about its corresponding heap cell, such as variables in which the cell is stored. Each edge in the heap graph is labeled with a field name; such labeled edges model which fields of cells point to other cells.

Example 1 *Figure 2.2 depicts distinct sets of graphs of heaps in* `alt_list` *states that (1) are alternating and (2) are not alternating (for now, ignore the pattern P with dashed nodes and edges in the center of Figure 2.2). The alternating heaps depicted consist of the alternating heaps with one to three non-nil cells. The non-alternating heaps depicted consist of three non-alternating heaps with one to two non-nil cells.*

Each pattern is a graph in which nodes and edges are labeled from the space of annotations as the labels on the nodes and edges of heap graphs. However, a pattern graph may also contain *summary nodes* and *definite* or *indefinite* labelings. Informally, a definite label indicates a fixed value (True or False), and an indefinite label indicates that either value is possible. A heap graph H is *matched* by a pattern graph P if there is a mapping h from the nodes of H to the nodes of P such that:

1. If multiple nodes of H are matched by a node n of P, then n is a summary node.

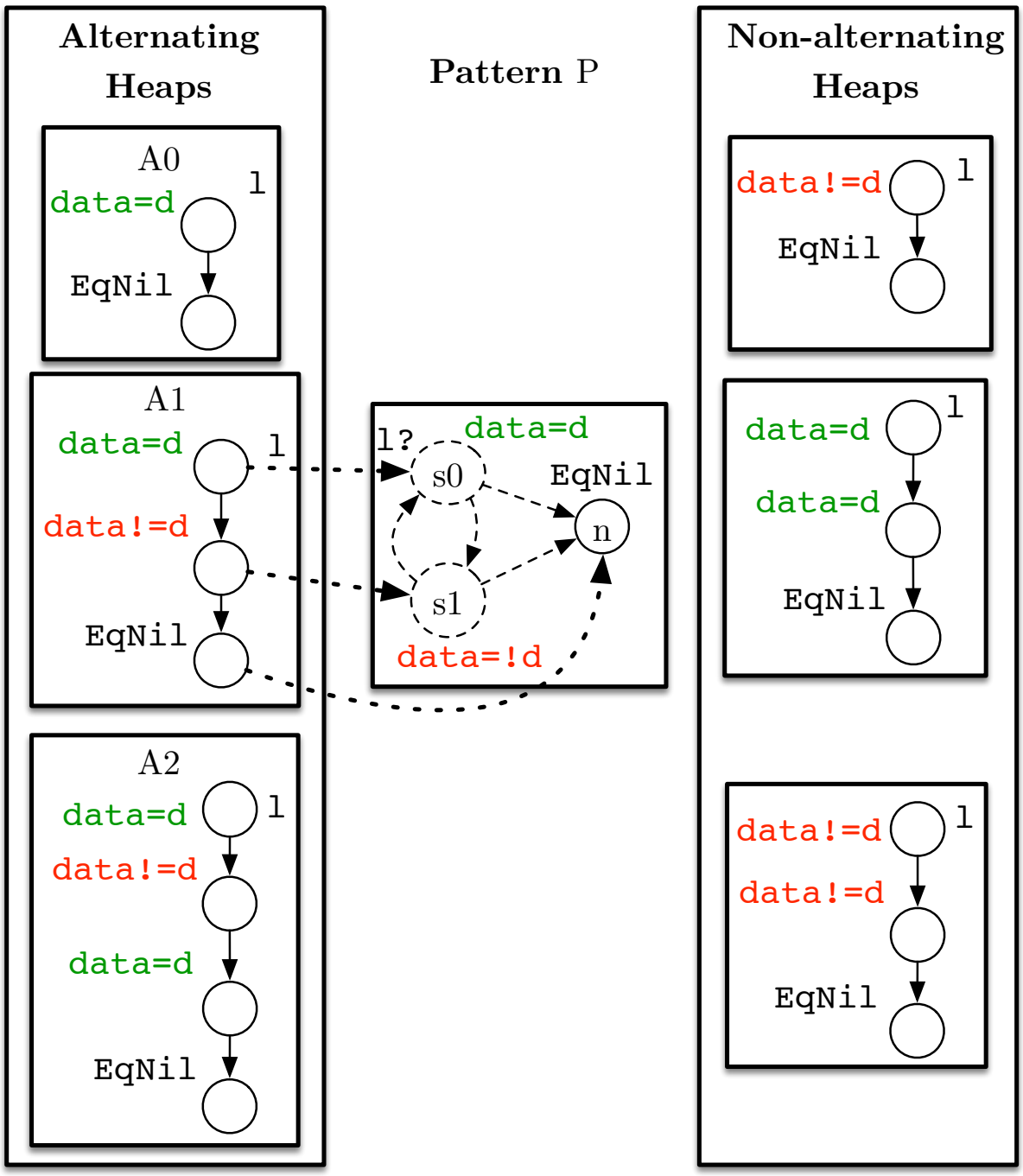

Figure 2.2: Sets of alternating and non-alternating heaps of `alt_list` (§2.1), depicted as graphs, and a heap pattern that matches each of the alternating and none of the non-alternating heaps. The labels of each node n are written adjacent to the node in a heap graph are written adjacent to n; some label values are colored for clarity. Each edge is implicitly labeled with the field name `next`. Each summary node of the pattern P is dashed, each indefinite node label is followed by a question mark, and each indefinite edge with an indefinite label is dashed. The dash-dotted lines from heap A_1 to P depict a matching from A_1 to P.

2. For each node n of H, if n is labeled with label L, then $h(n)$ is either definitely or indefinitely labeled with L. If n is not labeled with label L, then $h(n)$ is either definitely not labeled or indefinitely labeled with L.

3. For all nodes m and n of H, if there is an edge from m to n labeled with field `f`, then there is an edge from $h(m)$ to $h(n)$ either definitely or indefinitely labeled with `f`. If there is no edge from m to n labeled with `f`, then either there is no edge from $h(m)$ to $h(n)$ labeled with `f` or there is an edge from $h(m)$ to $h(n)$ indefinitely labeled with `f`.

Example 2 *The heap pattern P depicted in Figure 2.2 matches exactly the alternating heaps of* `alt_list`*. A matching from the second alternating heap A_1 in Figure 2.2 to P is depicted with dotted edges from the nodes of the heap to the nodes of the pattern. Because P matches each of the alternating heaps and none of the non-alternating heaps in Figure 2.2, we say that P* distinguishes *the alternating and non-alternating heaps.*

2.3 A proof structure for low-level heap programs

`alt_list` demonstrates that proving that some programs satisfy each of their assertions, which are defined purely over local variables, may still sometimes require a verifier to find invariants over the entire structure of the program heap. In particular, in order to prove that `alt_list` always satisfies its assertion at a line 12, a verifier must prove that at line 5, the heap always holds an alternating list.

PROVEIT attempts to construct proofs that are represented as a partial prefix-tree (i.e., an *unwinding tree*) of the program's control paths [McM06]. Each node in the tree represents an occurrence of a control location in a program path, and each edge in the tree represents a step of execution of the program over a sequence of non-branching instructions. Each node n is thus identified by a sequence of instructions $\texttt{Instrs}_n$ that the program must execute to reach the node, and is annotated with an *invariant*, represented as a heap pattern (§2.2) that is satisfied by all states reached after the program executes $\texttt{Instrs}_n$.

A program unwinding T is a proof that a given error location l_e is unreachable in any run of a program $\mathcal{A}$ if:

1. The initial heap of $\mathcal{A}$ satisfies the invariant at the root of T.

2. For each edge (m, n) in T, the invariant at m and the semantics of the instructions modeled by the edge (m, n) imply the invariant at n.

3. Each node in T that represents l_e is annotated with an invariant that is not satisfied by any program state.

4. Each leaf l that represents control location `L` of T either represents a terminal control location of P, or is *covered* by another node of T that represents `L`, and is annotated with a weaker invariant than the invariant of l.

If a leaf l is covered by node n, the intuitively the proof tree need not be further expanded from l, because any proof tree rooted at n, which is a proof of safety for all paths with prefix $\texttt{Instrs}_n$, is a valid proof of safety for all paths with prefix $\texttt{Instrs}_l$.

Example 3 *Figure 2.3 depicts a prefix tree T of an unwinding tree that proves that* `alt_list` *always satisfies the assertion at line 12. T models runs of* `alt_list` *that execute* `LOOP-CONS` *most three times. Nodes* 0, 2, 4, *and* 6 *model states of* `alt_list` *when control is at the loop head, and nodes* 1, 3, *and* 5 *model states of* `alt_list` *when control exits* `LOOP-CONS`*. Each edge of T is annotated with the sequence of instructions in* `alt_list` *that it models. Each node n of T is annotated with a heap pattern (§2.2) that over-approximates the set of heaps reachable by executing $\texttt{Instrs}_n$. Node* 6 *is covered by node* 4 *because the pattern at node* 6 *is entailed by the pattern at node* 4*. Thus, tree T can be expanded into a complete tree that proves the safety of P by expanding T from only leaf nodes* 1, 3, *and* 5*, not from leaf node* 6*.*

2.4 Learning heap patterns as a game

The key observation behind the design of PROVEIT is that any pair of disjoint sets of heaps H^+ and H^- defines a natural game, in which the objective for one player is to learn a pattern that distinguishes H^+ from H^-. In particular, let an *inductive pattern synthesizer* be an Oracle that takes as input a finite set of positive heaps H^+ and negative heaps H^-, and returns a heap pattern that is

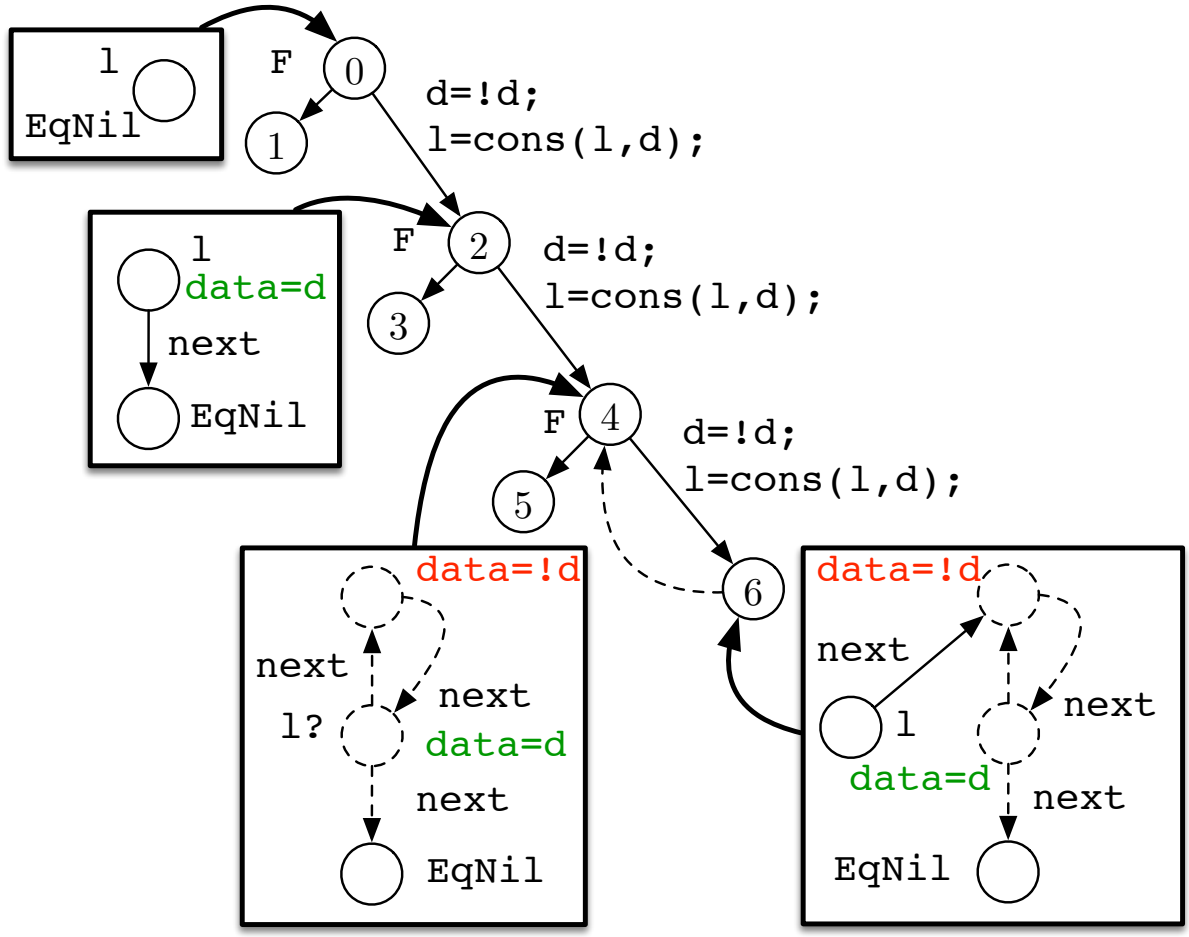

Figure 2.3: The prefix of an unwinding tree T that proves that no run of `alt_list` violates its assertion. Edge tree edge is annotated with the instruction sequence that it simulates, and each node is annotated with its pattern invariant.

matched by each heap H^+ and no heap in H^-. The synthesizer's goal in the game is to synthesize a heap pattern that distinguishes H^+ and H^-, without direct access to H^+ and H^-. The state of the game consists of finite subsets of *revealed* $H_0^+ \subseteq H^+$ and revealed $H_0^- \subseteq H^-$, and a pattern P generated by the synthesizer that distinguishes H_0^+ from H_0^-. In each play, if the inductive pattern synthesizer has not won, then the pattern verifier reveals either a heap in H^+ not matched P or a heap in H^- that is matched by P. The inductive pattern synthesizer then generates a new pattern that must match all revealed patterns in H^+ and no revealed patterns in H^-.

Example 4 *For* `alt_list`*, the reachable heaps and heaps that lead to an assertion violation from line* G_5 *(i.e., the alternating heaps and non-alternating heaps) define a pattern-synthesis game* G_5*. One valid play of the game with an inductive pattern synthesizer* O_5 *from a game state consisting of no revealed positive heaps and the non-alternating heaps in Figure 2.2 as revealed negative heaps is as follows: (1)* O_5 *plays the pattern that annotates node* 0 *in Figure 2.3; (2) the pattern verifier reveals heap* A_1 *in Figure 2.2; (3)* O_5 *plays the pattern that annotates node* 2 *in Figure 2.3; (4) the pattern verifier reveals heap* A_2 *in Figure 2.2; (5)* O_5 *plays the pattern that annotates node* 4 *in Figure 2.3 (i.e., pattern* P *in Figure 2.2), and wins the game.*

2.5 From game strategies to program proofs

The main result of our work is that while the problem of verifying heap-manipulating programs is undecidable, it can be reduced to winning a finite set of pattern-synthesis games. The primary difficulty in constructing an unwinding tree T that proves the safety of a program is in inferring patterns for the nodes of T that are (1) sufficiently strong enough to prove that each path of T to an error node is infeasible but (2) sufficiently weak that they can cover the patterns of sufficiently many leaf nodes to bound the set of program paths that must be modeled. For the example of `alt_list`, the pattern on node 4 of the unwinding tree in Figure 2.3 satisfies both of these criteria; however, in general the problem of inferring sufficient patterns is undecidable, and to date, there are not general-purpose analyses that can infer such invariants for practical programs.

In settings where it is not critical to infer properties of the program's entire heap, it is sufficient to infer invariants for an unwinding tree by obtaining an *interpolant* [McM06] for each node n of

two formulas describing (1) states reachable by executing the instruction sequence of n from the beginning of the program and (2) states from which n reaches an error. However, interpolation-based approaches must infer interpolants as formulas in a theory for which the problem of constructing an interpolant is decidable, typically the combined theory of linear arithmetic, uninterpreted functions, and arrays (LIUFA). Unfortunately, formulas in such theories cannot naturally describe sets of heaps with arbitrarily many cells, such as the set of alternating heaps of `alt_list` (§2.1).

In this work, we explore a framework in which a verifier can learn heap patterns that are sufficient invariants for proving the correctness of a program, under the assumption that the verifier can query an oracle that can efficiently win a class of the heap-learning games described in §2.4.

Example 5 *Suppose that* PROVEIT *has constructed the unwinding tree depicted in Figure 2.3, and must choose a pattern with which to annotate node* 4*, which models line 5 of* `alt_list`*.* PROVEIT *could choose a pattern* P *that only distinguishes between alternating and non-alternating lists of length less than or equal to two. However,* P *is, intuitively too strong an invariant in that it cannot cover any valid annotation of node* 6*.*

Alternatively, if PROVEIT *plays the game* G_5 *as pattern verifier against the pattern synthesizer* O_5 *(Ex. 4), and annotates node* 4 *with the winning pattern played by* O_5 *(as depicted in Figure 2.3), then* PROVEIT *can construct an unwinding tree that proves the safety of* `alt_list`*.*

The main result that we present in this thesis is that for a given program P, if our program verifier PROVEIT has access to an inductive pattern synthesizer that wins a game defined analogously to the game G_5 defined for line 5 of Figure 2.1 (Ex. 4) for each control location of P, then PROVEIT verifies P successfully.

Summary

In this chapter, we presented a brief overview of the problem domain, and our approach. Our algorithm extends interpolation-based verification techniques to heap-manipulating programs, using an Oracle to provide input for the interpolation step. The next chapter will focus on technical

background for modeling programs for verification.

Chapter 3

Background

In this section, we define the formalize requirements for lazy interpolation-based model checking. This chapter is entirely based on [McM06]. This applies to the standard domain of model checking for programs that don't use heap memory, 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 Σ of non-logical symbols. For a given formula or set of formulas ϕ we will use $\mathcal{L}(\phi)$ to denote *wffs* over the vocabulary of ϕ.

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 ϕ, we will use the notation $\phi^{\langle n \rangle}$ to denote the addition of n primes to every symbol in ϕ (meaning ϕ at n time units in the future). For any set Σ of symbols, let Σ' denote $\{s' | s \in \Sigma\}$ and $\Sigma^{\langle n \rangle}$ denote $\{s^{\langle n \rangle} | s \in \Sigma\}$.

3.1 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 Λ is a finite set of program locations, Δ 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 $path$ π 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 π 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 π 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 to 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.

3.2 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 [Cra57] 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 Γ 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 Γ is quantifier-free, we can derive a quantifier-free interpolant for Γ from a refutation of Γ, in certain interpreted theories [McM05].

3.3 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 ϵ, $M_v : V \to \Lambda$ 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* $\psi(v) \equiv \mathsf{False}$*. 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:*

- $\psi(\epsilon) \equiv \mathsf{True}$*, 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*

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 [McM06].

3.4 Impact Algorithm

This section describes a semi-algorithm from [McM06], 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)

Each of the three steps preserves well-labeledness of the unwinding. 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.

3.4.1 Algorithm Description

We now briefly explain how the three main procedures EXPAND, REFINE, and COVER work.

EXPAND, formally outlined in Algorithm 1, explores new actions for an uncovered leaf. An uncovered leaf (described in Defn. 2) is intuitively one for which all possible successor states have not been explored yet. If the vertex $v \in V$ supplied to EXPAND is an uncovered leaf, then for each possible action T at the program location $M_v(v)$, a new vertex w is added to the unwinding tree (line 4). This is normally done when a leaf is such that it cannot be covered by any other vertex in the tree, and thus needs further state space exploration.

REFINE is the procedure that generates the interpolant, and can potentially mark an error vertex unreachable. Firstly, we note that if REFINE succeeds, then $\phi(v)$ must be False (since $\hat{A}_n$ is always False). Thus, to make the unwinding safe, we have to only apply REFINE to every error vertex (line 2) that hasn't yet been marked False. For such an error vertex v, REFINE queries a decision procedure [McM05] to find an interpolant for the path from root to error vertex (line 4). If an interpolant is not found, it means that the error state is reachable, and the program is marked unsafe (line 13). If an interpolant is found, then it is used to strengthen the predicate labelings at the vertices along the path π. In particular, if the interpolant formula at a vertex v_i doesn't subsume the current

```
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.

predicate label at v_i (line 7), then the current label can be strengthened. Note that if this happens, then the stronger label at v_i may possibly lead it to stop covering some of the vertices it previously covered. This is taken into account by removing all pairs $(\cdot, v_i)$ in the covering relation $\triangleright$ (line 8).

COVER is a simple procedure that takes two vertices $v, w \in V$, and attempts to cover v with w, provided v is an uncovered vertex that is not an ancestor of w, and they are associated with the same program location (line 2). If the predicate label at v is subsumed by the predicate label at w, then it means that v need not be explored further, and can be covered with w, by adding the pair (v, w) to $\triangleright$. Note that if a vertex is covered, then all its descendants are also automatically covered. Since we don't allow a covered vertex to cover other vertices, each vertex y which is a descendant of v, and covers other vertices, must stop covering those vertices (line 5).

3.4.2 Termination

The Impact algorithm works by repeated applications of EXPAND, COVER, and REFINE. When any of the three procedures don't cause any change in the unwinding tree, the algorithm terminates. To build a well-labeled unwinding, a strategy is required for applying the three procedures above. The most difficult question is when to apply COVER. Covering one vertex can result in uncovering others. Thus, applying COVER non-deterministically may not terminate.

To avoid the possibility of of non-termination, a total order $\preceq$ can be defined on the vertices.

```
1  Procedure REFINE(v ∈ V):
2      if M_v(v) = l_f and ψ(v) ≢ False 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          if U(π) has an interpolant Â_0, ··· , Â_n then
5              for i = 0 ··· n do
6                  let φ = Â_i^{⟨−i⟩}
7                  if ψ(v_i) ⊭ φ then
8                      remove all pairs (·, v_i) from ▷;
9                      set ψ(v_i) ← ψ(v_i) ∧ φ;
10                 end
11             end
12         else
13             abort (program is unsafe)
14         end
15     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.

This order must respect the ancestor relation. That is, if $v \sqsubset w$, then $v \prec w$. For example, we could define $\preceq$ by a pre-order traversal of the tree, or numbering the vertices in order of creation. After that, COVER is restricted to only those pairs (v, w) for which $w \prec v$. If adding (v, w) to $\rhd$ leads to the removal of (x, y) where $v \sqsubseteq y$, then by transitivity, we must have $v \prec x$. Thus, covering a vertex v can only result in uncovering vertices greater than v, implying that we cannot apply COVER indefinitely.

We call a vertex v *closed* if either it is covered, or no arc (v, w) can be added to $\rhd$ while maintaining well-labeledness. It is possible to guarantee that when a vertex is expanded, all of its ancestors are closed, thus making it so that a vertex that could be covered isn't expanded unnecessarily. The UNWIND algorithm in [McM06] describes a systematic way of doing this. Forced covering (calling COVER on all nodes of a path after a REFINE step) also serves as an optimization. All of these ideas apply directly to the case of heap-manipulating programs.

```
1 Procedure COVER(v, w ∈ V):
2     if v is uncovered and M_v(v) = M_v(w) and v ⊭ w then
3         if ψ(v) ⊨ ψ(w) then
4             add (v, w) to ▷;
5             delete all (x, y) ∈ ▷, s.t. v ⊑ y;
6         end
7     end
```

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

Summary

In this chapter, we presented a framework for modeling programs as state graphs, and an interpolant-based approach to verification that uses program unwindings to explore the state space. In the next chapter, we describe the formalism that allows us to represent and reason about heap memory. This will be useful for defining the verification algorithm for heap-manipulating programs.

Chapter 4

Heap Patterns

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 [SRW02], which we use to formulate patterns over program heaps (§4.2).

4.1 Language Definition

In this section, we define the syntax (§4.1.1) and semantics (§4.1.2) of our subject language LANG.

4.1.1 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 `Locs`, predicate variables $\texttt{Vars}_P$, heap variables $\texttt{Vars}_H$, and heap fields `Fields`. A program is a sequence of instruction, each labeled with a control location (Equation 4.1). An

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

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

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

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

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

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

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

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

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

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.

instruction either updates the program's predicate variables or heap variables (Equation 4.2). An instruction that updates a 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).

4.1.2 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.

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

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

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

4.2 Heap pattern Language

Definition 4 *A* LANG heap *is a tuple* $(C, V_H, F, \mathsf{PredLbl})$*, 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. $\mathsf{PredLbl} : C \times \mathtt{Vars}_P \to \mathbb{B}$ *defines an assignment for each cell* $c \in C$ *and predicate in* $\mathtt{Vars}_P$ *to* True *or* False*. We denote the space of predicate labeling functions as* $\mathsf{PredLblFn} = \mathcal{C} \times \mathtt{Vars}_P \to \mathbb{B}$*.*

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 [SRW02].

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 $\mathsf{Pats} = \mathcal{P}(\mathcal{N}) \times \mathsf{VarLbls} \times \mathsf{PredLbls} \times \mathsf{LblEdges}$.

Additionally, we define the universal and empty patterns (1_D and 0_D respectively). These are analogous to True and False.

Program in a language with low-level memory updates, such as `alt_list` (§2.1), 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 6 *Pattern* P *(§2.2, Figure 2.2) 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* $h = (C, V_H, F, \mathsf{PredLbl})$, *the* concrete pattern *of* h *is* $G_h = (N_h, \mathtt{V}_h, \mathtt{P}_h, E_h, \sigma_h)$, *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* $\mathtt{V}_h(c, \mathtt{p}) = \mathsf{True}$, *and otherwise,* $\mathtt{V}_h(c, \mathtt{p}) = \mathsf{False}$.
3. *The predicate labeling carries over directly, i.e.* $\mathtt{P}_h(n, p) = \mathsf{PredLbl}(c, p)$ *where* $p \in \mathtt{Vars}_P$, *and* $n \in N_h$ *corresponds to* $c \in C$.
4. *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* $E_h(c, \mathtt{f}, c') = \mathsf{True}$, *and otherwise,* $E_h(c, \mathtt{f}, c') = \mathsf{False}$.
5. $\sigma_h(n) = \mathsf{False}$ *for every* $n \in N_h$.

Example 7 *§2.2, Figure 2.2 contains concrete patterns for six heaps: three alternating heaps and three non-alternating heaps.*

4.3 Pattern Entailment

The entailment relation over heap patterns formulates 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 7 *Let the* information-precision ordering $\sqsubseteq_3 \subseteq \mathbb{B}_3 \times \mathbb{B}_3$ *be such that* True $\sqsubseteq_3$ Maybe *and* False $\sqsubseteq_3$ Maybe.

For all heap patterns $P, P' \in$ Pats *with* $P = (N, \mathtt{V}, \mathtt{P}, E, \sigma)$ *and* $P' = (N', \mathtt{V}', \mathtt{P}, E', \sigma')$, P entails P', *denoted* $P \models P'$, *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 $h \in$ Heaps and pattern $P \in$ 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 $P_0, P_1 \in$ Pats and heap $h \in$ Heaps, if $P_0 \models P_1$ and $h \preceq P_0$, then $h \preceq P_1$.

Example 8 *In §2.2, Figure 2.2, 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.

By definition, every pattern $P \neq 0_D$ satisfies the following properties:

- $P \models 1_D$
- $P \not\models 0_D$
- $0_D \models P$

4.4 Modeling Heap Programs

We can now use the heap patterns described in Defn. 5 to model programs for our modified Impact algorithm for heap-manipulating programs. Heap programs are modeled similarly to other programs, as described in §3.1. For the purposes of the Impact algorithm, the key difference is in

how program unwindings are constructed. Instead of labeling vertices of the unwinding with FOL formulas, we use heap patterns.

Definition 8 *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 $\mathcal{A}$*
- $\psi : V \to \mathsf{Pats}$ *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.

Each heap pattern contains a special node `NIL`, which is used to represent the location of pointers that aren't allocated to a specific node.

Summary

In this chapter, we presented a language LANG for imperative programs that perform low-level heap operations. We defined a formal representation of a heap for LANG, and the idea of heap patterns which can be used to represent multiple heaps. We discussed the idea of pattern entailment, and the modeling of LANG programs for verification. The next chapter describes PROVEIT, our algorithm for verifying heap-manipulating programs.

Chapter 5

PROVEIT: The Heap Impact Algorithm

Building on top of the framework defined in §3.4 and §4, we can define PROVEIT, by modifying the Impact algorithm to work for heap-manipulating programs. In this chapter, we first define the three steps of Impact , that is EXPAND, COVER, and REFINE for PROVEIT, respectively calling them EXPANDP, COVERP, and REFINEP. Then we describe an interpolant-learning procedure that retrieves patterns from an Oracle, thereby completing the description of the algorithm.

5.1 Notation Review

In this section, we briefly go over some of the notation from previous chapters that will be relevant to the description of PROVEIT.

5.1.1 Programs

Recall that a *program* is a tuple $(\Lambda, \Delta, l_i, l_f)$, where Λ is a finite set of program locations, Δ 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.

5.1.2 Program Paths

A *path* π 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 π 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.

For further details, the reader is referred to §3.1.

5.1.3 Program Unwindings

As defined in Defn. 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 ϵ, $M_v : V \to \Lambda$ is the vertex map, and $M_e : E \to \Delta$ is the edge map, such that $M_v(\epsilon) = l_i$. Also, 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 $M_e(v, w) = T$. Intuitively, an unwinding is a tree where each vertex is mapped (M_v) to a program location, and each edge is mapped (M_e) to a step (action) in the program.

As defined in Defn. 8, 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 $\mathcal{A}$, $\psi : V \to \mathsf{Pats}$ is called the vertex labeling, and $\rhd \subseteq V \times V$ is the covering relation. Intuitively, a labeled unwinding is a program unwinding where each vertex is labeled (ψ) with a heap pattern, and a covering relation $\rhd$ is used, which is used to keep track of covered vertices that need not be explored further.

5.1.4 Postcondition Transforms for Heap Operations

In addition to the previous definitions, we define the operator Post, which will be used for generating examples to interact with the Oracle. Post is an operator for computing strongest postconditions for heap patterns modified by LANG instructions.

Definition 9 *The operator* Post*, which computes the strongest postcondition for a given heap pattern, and action. That is* $\mathsf{Post} : \mathsf{Pats} \times \mathcal{T} \rightarrow \mathsf{Pats}$*, where* $\mathcal{T}$ *is the set of all actions.*

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

We also note that the Post operator can be overloaded to work with individual heaps instead of patterns, since single heaps can also be represented using a pattern.

The formal rules for computing Post for each individual action are presented here. We define them for major heap operations that are part of LANG. The formal requirement is that for a heap h, pattern P, and transition T, such that $h \preceq P$, we must have $\mathsf{Post}(h, T) \preceq \mathsf{Post}(P, T)$. Assume that the original pattern is represented by $P = (N, \mathtt{V}, \mathtt{P}, E, \sigma)$, and the pattern after transformation by Post is represented by $P' = (N', \mathtt{V}', \mathtt{P}', E', \sigma')$. We define P' using the definition of P, for each possible value of T below. New variables are presented as updates to the values of old variables.

- **ALLOC** ($v := alloc()$):
  - $N' = N \cup \{n\}$ where $n \notin N$ is a new node allocated by $alloc()$
  - $\mathtt{V}'$ updates $\mathtt{V}$ such that $\mathtt{V}'(n, v) = \mathsf{True}$ and $\forall m \neq n, \mathtt{V}'(m, v) = \mathsf{False}$
  - $\mathtt{P}'$ updates $\mathtt{P}$ such that $\forall p, \mathtt{P}'(n, p) = \mathsf{Maybe}$
  - E' updates E such that $E'(n, f, m) = \mathsf{False}$, and $E'(m, f, n) = \mathsf{False}$ for all fields f and nodes $m \neq n$
  - σ' updates σ such that $\sigma'(n) = \mathsf{True}$
- **COPY** ($v1 := v2$):
  - $N' = N$
  - $\mathtt{V}'$ updates $\mathtt{V}$ such that $\forall n \in N, \mathtt{V}'(v1, n) = \mathtt{V}(v2, n)$
  - $\mathtt{P}' = \mathtt{P}$
  - $E' = E$
  - $\sigma' = \sigma$
- **LOAD** ($v1 := v2 \rightarrow f$):
  - $N' = N$

- $\mathtt{V}'$ updates $\mathtt{V}$ as follows:

  Let $S = \{n \in N : \mathtt{V}(n, v2) = \mathsf{True} \vee \mathtt{V}(n, v2) = \mathsf{Maybe}\}$

  Let $T = \{n \in N : \exists s \in S \cdot E(s, f, n) = \mathsf{True} \vee E(s, f, n) = \mathsf{Maybe}\}$

  if $T = \{t\}$ (singleton), then $\mathtt{V}'(t, v1) = \mathsf{True}$, otherwise $\forall t \in T, \mathtt{V}'(t, v1) = \mathsf{Maybe}$

- $\mathtt{P}' = \mathtt{P}$

- $E' = E$

- $\sigma' = \sigma$

- **STORE** ($v1 \rightarrow f := v2$):

- $N' = N$

- $\mathtt{V}' = \mathtt{V}$

- $\mathtt{P}' = \mathtt{P}$

- E' updates E as follows:

  Let $S = \{n \in N : \mathtt{V}(n, v1) = \mathsf{True} \vee \mathtt{V}(n, v1) = \mathsf{Maybe}\}$

  Let $T = \{n \in N : \mathtt{V}(n, v2) = \mathsf{True} \vee \mathtt{V}(n, v2) = \mathsf{Maybe}\}$

  $\forall s \in S, t \in T, E'(s, f, t) = \mathsf{Maybe}$ (and True if both S and T are singletons)

- $\sigma' = \sigma$

- **PREDICATE** (instr_P):

- $N' = N$

- $\mathtt{V}' = \mathtt{V}$

- $\forall n \in N, p \in \mathtt{Vars}_P, \mathtt{P}'(n, p) = post(p, \mathtt{P}(n, p), \mathsf{instr}_P)$, where $post$ is a postcondition operator for three-valued predicates

- $E' = E$

- $\sigma' = \sigma$

5.1.5 Interpolants for Heap Programs

Based on §3.2, we can define heap pattern interpolants for a sequence of path formulas $\Gamma = A_1, \cdots, A_n$. We say that $\hat{P}_0, \cdots, \hat{P}_n$ is an *interpolant* for Γ when

- $\hat{P}_0 = 1_D$ and $\hat{P}_n = 0_D$ and,

- for all $1 \leq i \leq n$, $\mathsf{Post}(\hat{P_{i-1}}, A_i) \models \hat{P}_i$ and

This means that the the Post transform (strongest postcondition) of $\hat{P_{i-1}}$ over A_i entails $\hat{P}_i$.

5.2 ProveIt

This section describes our algorithm for building a complete, safe, well-labeled unwinding of a heap-manipulating program. Our algorithm builds on top of the Impact algorithm defined in §3.4. Our algorithm may not terminate if the program is safe. If the program is unsafe, and our Oracle can help find the right interpolant, then our algorithm will terminate. A non-deterministic procedure with three basic steps is outlined here. The three steps are

- ExpandP, which generates the successors of a leaf vertex (Algorithm 4)
- RefineP, which refines the labels along a path, labeling an error vertex False (Algorithm 5)
- CoverP, which expands the covering relation (Algorithm 6)

Each of the three steps preserves well-labeledness of the unwinding. 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.

In addition to the three steps above, there are two main procedures that are needed by RefineP which allow it to interact with the Oracle and generate interpolants for the verification. The procedures are

- InterpLearner, which RefineP relies on to provide a valid interpolant for an error location (Algorithm 7)
- NewCandidate, which is used by InterpLearner to interact with the Oracle, and query for a heap pattern given positive and negative concrete heaps (Algorithm 9)

5.2.1 Algorithm Description

We now briefly explain how the main procedures for PROVEIT work. The outline is very similar to that of the Impact algorithm defined in §3.4, but our modifications make it possible to use heap patterns in place of simple FOL formulas.

EXPANDP, formally outlined in Algorithm 4, explores new actions for an uncovered leaf. An uncovered leaf (described in Defn. 2) is intuitively one for which all possible successor states have not been explored yet. If the vertex $v \in V$ supplied to EXPANDP is an uncovered leaf, then for each possible action T at the program location $M_v(v)$, a new vertex w is added to the unwinding tree (line 4). This is normally done when a leaf is such that it cannot be covered by any other vertex in the tree, and thus needs further state space exploration. Notice that this is identical to Impact , but with they key difference that a new vertex in the unwinding is initialized with the universal heap 1_D instead of True (line 5).

1 **Procedure** EXPANDP($v \in V$):
2 **if** *v is an uncovered leaf* **then**
3 **foreach** *action* $(M_v(v), T, m) \in \Delta$ **do**
4 add a new vertex w to V and a new edge (v, w) to E;
5 set $M_v(w) \leftarrow m$ and $\psi(w) \leftarrow 1_D$;
6 set $M_e(v, w) \leftarrow T$;
7 **end**
8 **end**

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

REFINEP is the procedure that generates the interpolant, and can potentially mark an error vertex unreachable. Firstly, we note that if REFINEP succeeds, then $\phi(v)$ must be 0_D, the empty heap pattern (since $\hat{A}_n$ is always 0_D). Thus, to make the unwinding safe, we have to only apply REFINEP to every error vertex (line 2) that hasn't yet been marked 0_D. For such an error vertex v, REFINEP makes a call to INTERPLEARNER (Algorithm 7) to find an interpolant for the path from root to error vertex (line 4). If an interpolant is not found it could mean one of two things

1. The program has a bug, and the error vertex is actually reachable

2. The interaction with the Oracle did not result in an interpolant being found

Since our Oracle cannot be guaranteed to definitely provide an interpolant, at this point, we quit REFINEP and try again later when more vertices have been labeled with stronger heap patterns (line 14). If an interpolant is found, then it is used to strengthen the heap pattern labelings at the vertices along the path π. In particular, if the interpolant formula at a vertex v_i is not entailed by the current heap pattern label at v_i (line 8), then the current label can be strengthened. Note that if this happens, then the stronger label at v_i may possibly lead it to stop covering some of the vertices it previously covered. This is taken into account by removing all pairs $(\cdot, v_i)$ in the covering relation $\rhd$ (line 9).

COVERP is a simple procedure that takes two vertices $v, w \in V$, and attempts to cover v with w, provided v is an uncovered vertex that is not an ancestor of w, and they are associated with the same program location (line 2). If the heap pattern label at v entails the heap pattern label at w, then it means that v need not be explored further, and can be covered with w, by adding the pair (v, w) to $\rhd$. Note that if a vertex is covered, then all its descendants are also automatically covered. Since we don't allow a covered vertex to cover other vertices, each vertex y which is a descendant of v, and covers other vertices, must stop covering those vertices (line 5).

Learning Invariants from Positive and Negative Examples

We now describe the algorithms that allow PROVEIT to interface with the Oracle and find invariants. In particular, the REFINEP procedure (Algorithm 5) queries INTERPLEARNER for an interpolant, which in turn queries NEWCANDIDATE (Algorithm 9). NEWCANDIDATE makes the call to the Oracle. We now describe these algorithms in greater detail.

INTERPLEARNER contructs the interpolant for an unfolding $\mathcal{U}(\pi)$ of path π, where π is a program path ending at an error location. Paths and unfoldings are formally defined in §3.1, while interpolants for path sequences in heap-manipulating programs are defined in §5.1.5. These definitions apply directly to our case of heap-manipulating programs. INTERPLEARNER maintains

```
1  Procedure REFINEP(v ∈ V):
2      if M_v(v) = l_f and ψ(v) ≢ 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 = INTERPLEARNER (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 (retry later)
15         end
16     end
```

Algorithm 5: REFINEP: takes as input a vertex $v \in V$ at an error location and tags the path from root to v with invariants. INTERPLEARNER (Algorithm 7) is the procedure that queries the Oracle and provides an interpolant that can be used in REFINEP. Notice that if the interpolant cannot be found, we might have to try later after the unwinding tree has changed.

two types of data:

1. A current set of candidate patterns $\hat{A} = \hat{A}_0, \hat{A}_1, \cdots, \hat{A}_n$ for the vertices of the unwinding tree associated with path π
2. A set of positive and negative candidates $H_i^+, H_i^- 0 \leq i \leq n$, which contain concrete heaps that serve as positive and negative examples respectively for the Oracle

If $\hat{A}$ is already an interpolant for $\mathcal{U}(\pi)$, it is returned. If not, then we pick some $\hat{A}_i$ for an update, which is done by calling NEWCANDIDATE (line 8). NEWCANDIDATE returns an updated canddiate pattern that is set to $\hat{A}_i$, and the procedure continues, until a full path interpolant is found.

1 **Procedure** `COVERP`$(v, w \in V)$*:*
2 **if** v *is uncovered and* $M_v(v) = M_v(w)$ *and* $v \not\models w$ **then**
3 **if** $\psi(v) \vDash \psi(w)$ **then**
4 add (v, w) to $\rhd$;
5 delete all $(x, y) \in \rhd$, s.t. $v \sqsubseteq y$;
6 **end**
7 **end**

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

NEWCANDIDATE also udpates the sets H_i^+, H_i^-, which we describe later. Note that the check for whether the current set of candidates are actually an interpolant is delegated to the ISINTERPOLANT procedure (line 6).

1 **Procedure** `INTERPLEARNER`$(\mathcal{U}(\pi))$*:*
2 Let $\pi = (l_0, T_0, l_1)(l_1, T_1, l_2) \cdots (l_{n-1}, T_{n-1}, l_n)$
3 Set $\hat{A}_i = 1_D, 0 \leq i < n, \hat{A}_n = 0_D$
4 Set $H_i^+ = \{\}, 0 \leq i \leq n$
5 Set $H_i^- = \{\}, 0 \leq i \leq n$
6 **while** $\neg$ISINTERPOLANT $(\hat{A}, \mathcal{U}(\pi))$ **do**
7 pick $i \in \{1, 2, \cdots, n-1\}$
8 $\hat{A}_i =$ NEWCANDIDATE$(l_i, \hat{A}, H^+, H^-, \mathcal{U}(\pi))$
9 **end**
10 **return** $\hat{A}_0, \hat{A}_1, \cdots, \hat{A}_n$

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

ISINTERPOLANT is a simpler procedure that checks if the current set of candidates $\hat{A}_0, \hat{A}_1, \cdots, \hat{A}_n$ satisfy the definition of path interpolants in §5.1.5.

NEWCANDIDATE takes in a program location l_i, the current set of candidates $\hat{A}$, positive and negative examples H_i^+, H_i^- seen so far, and the unfolding $\mathcal{U}(\pi)$. It returns a candidate can be used to update an existing candidate, while also updating the sets H_i^+, H_i^-. NEWCANDIDATE is the core procedure, as far as interacting with the Oracle is concerned. Here, we describe in simple terms

```
1 Procedure ISINTERPOLANT(Â, 𝒰(π)):
2     if Â_0, Â_1, ··· , Â_n is an interpolant for 𝒰(π) then
3         return True
4     end
5     return False
```

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

how it works for a given location l_i in a program path π:

1. Compute the strongest postcondition S, starting at the root of the tree (which is annotated with the universal pattern 1_D) until l_i. This partial path is indicated by $\pi_{0,i}$ (line 3). S will be useful to generate concrete heaps later on. We also start with an initial value of 1_D for the candidate C.

2. Inside a loop to query the Oracle $\mathcal{O}$ for a heap pattern (line 6). The Oracle uses the current sets of positive and negative examples of concrete heaps (H_i^+, H_i^- respectively).

3. The candidate C at a location must be entailed by the strongest postcondition S at that location. If this does not happen, it means that the candidate is not "weak enough". In other words, there is at least one concrete heap h in S that is not contained in C. To indicate this, h is added to the set of positive examples H_i^+ (line 8), and the Oracle is queried again for an updated pattern candidate.

4. If the Post transform of the candidate C across the edge with action T_i does not entail the candidate $\hat{A}_{i+1}$ at the next vertex in the path (line 11), then it means that the required condition for interpolation that we defined in §5.1.5 is not satisfied. This means that C is "strong enough", that is, it contains a heap h that does not satisfy the interpolant condition, and should be removed. This heap is added to the set of negative examples H_i^+ (line 13), and the Oracle is queried again for an updated pattern candidate.

5. Finally, if none of these conditions hold, the candidate is returned to INTERPLEARNER, where it is incorporated into the current set of candidates for the interpolant.

Procedure NEWCANDIDATE($l_i, \hat{A}, H^+, H^-, \mathcal{U}(\pi)$):

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**

  $C = \mathcal{O}(H_i^+, H_i^-)$

  if $S \not\models C$ **then**

    $H_i^+ = \{h\} \cup H_i^+$ where $h \in S, h \notin C$

    continue

  end

  if $\mathsf{Post}(C, T_i) \not\models \hat{A}_{i+1}$ **then**

    $\exists h \cdot h \preceq C \wedge \mathsf{Post}(h, T_i) \not\preceq \hat{A}_{i+1}$

    $H_i^- = \{h\} \cup H_i^-$

    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), and unfolding $\mathcal{U}(\pi)$ of path π, and interacts with the Oracle $\mathcal{O}$ to find a new candidate for l_i.

Summary

This chapter presented PROVEIT, our algorithm for verifying heap-manipulating programs. PROVEIT uses an Oracle to perform the interpolation step, which helps in learning heap patterns from positive and negative concrete examples. The next chapter describes the implementation of an interface for one such Oracle - a human user.

Chapter 6

User Interface to a Human Oracle

In this chapter, we describe one implementation of the Oracle. In Algorithm 9, the candidate is obtained as $C = \mathcal{O}(H_i^+, H_i^-)$. The Oracle $\mathcal{O}$ considers two sets of concrete heaps H_i^+ and H_i^-, and returns a candidate pattern C which is then evaluated. After the evaluation, it is either accepted, or needs to be further refined, in which case H_i^+ or H_i^- are updated, and a new query is submitted to the Oracle. This process repeats until a suitable candidate is found for the current vertex in the program unwinding.

6.1 Human as an Oracle

Human beings are good at finding patterns in several types of spatial data. This fact has been utilized for crowdsourcing the harder parts of several difficult technical problems [LSJ12, ver14, eye12]. The important challenges are to identify steps in the problem where human insight is critical, find ways to transform these steps into tasks that non-expert humans can perform, and combine the results to resolve those steps in the solution. In our problem of verifying heap-manipulating programs, the goal of the Oracle is to provide heap patterns that can be used to generate interpolants, which are then used for verification. Finding these interpolants automatically is a difficult step in program verification, while checking a provided interpolant for correctness is relatively easier to automate. Later in §6.4.1 and §6.4.2, we discuss more elaborately how our heap pattern language

and interface fare as tools to input human insight.

6.2 Web Interface

As a practical demonstration of the Oracle, we designed a web interface that shows examples of concrete heap to a human user. The interface allows them to construct a pattern graph using a graphical interface. The pattern graph should be such that it covers all positive examples, but doesn't cover any negative example.

Our graphical interface, created using D3 [d3j16] is a way for our Oracle (human) to interact with the PROVEIT algorithm. The interface is designed to make it simple to input heap pattern graphs and receive quick feedback about the correctness the pattern graph. Pattern graphs have several characteristics, as described in Defn. 5, and it can be challenging and overwhelming for the user to input a graph that generalizes enough, but not so much that it becomes useless for the underlying verification algorithm. Since our Oracles are humans, it is crucial that we create the best experience for them to input information to the prover.

Later, we describe in §6.2.2 what kind of background we expect from our users. One of our goals was to design an interface that would not need expertise in software verification. This means that the user input language has to be such that it not only works well with our heap pattern formalism, but also avoids the need for users to interact with the formalism directly. Since heaps and heap patterns are essentially graphs, we decided it was reasonable to represent these as visual graphs, and also allow users to "draw" graphs while getting instant feedback. We decided to use force-directed graphs in D3 for our implementation.

6.2.1 Force-Directed Graphs in D3.js

Before moving on with the description of our implementation, we briefly describe the Javascript library D3 that we used to build our heap pattern web interface. D3 allows to bind data to a Document Object Model (DOM), and then apply data-driven transformations to the document. For

example, D3 can be used to generate an HTML table from an array of numbers. Or, the same data can be used to create an interactive SVG bar chart with smooth transitions and interaction. The key problem solved by D3 is efficient manipulation of documents based on data. D3 is capable of supporting large datasets, and a wide variety of dynamic behaviors and interactions.

Force-directed graph drawing algorithms [Ead84, FR91] are a class of algorithms for drawing graphs in an aesthetically pleasing way. Their purpose is to position the nodes of a graph in two or three-dimensional space so that all edges are of more or less equal length and there are as few crossing edges as possible, by assigning forces among the set of edges and the set of nodes, based on their relative positions, and then using these forces either to simulate the motion of the edges and nodes or to minimize their energy. This is exactly like a simulation of an actual physical system in space. It is possible to define different "forces" such as those arising from gravity, electrical charges, elasticity, etc., and assign them to edges, nodes, and other components of the graph. The algorithm combines all these forces together, trying to reach a stable state. The key is to carefully calibrate the forces so that the resulting graph visualization is suitable. There are several ways force-directed graphs can be modeled, but we used D3's inbuilt model, that makes it really easy to create a simple force-directed graph. It works by defining nodes and links, and appends HTML objects to them. As the nodes and links interact to reach stable state, the HTML objects move along, making the visualization a graph built on top of an underlying force-directed graph drawing algorithm.

Advantages and Disadvantages of Force-directed graphs

We chose force-directed graphs because they are designed for drawing graphs in an aesthetically pleasing way, especially ones for which structure is unknown beforehand. Our positive and negative heap examples are often large and have diverse structures, and it's not trivial to have a pre-defined way of drawing them. Force-directed drawing algorithms take care of this, and are highly configurable. Our earlier familiarity with them was also a factor in the choice.

The one major disadvantage is that the spatial relationships of the original nodes change when nodes or edges are modified. Essentially, a local modification can cause a global change of layout, making it harder for users to remember how the same example changed over time. In the future, it will certainly be worthwhile to explore more graph-drawing algorithms.

6.2.2 Interface for Drawing Pattern Graphs

The basic interface for drawing heap graphs is shown in Figure 6.1. This is what it would look like on starting the visualization. The graph on the right is a force-directed graph representing the current heap pattern. Nodes are represented by circles, and edges by arrows connecting the circles. Arrows are directed, and for simplicity we assume that each node has outgoing fields of only one type. For instance, a data structure with a $next$ pointer would fit this case very well. All nodes and edges are interactive, and allow for different actions to express a rich language of heap patterns.

The form on the left allows the user to interact with the graph and set values for the heap and predicate labelings. The predicates and heap variables have been inferred by the program analysis already.

Example 9 *In Figure 6.1, the user can select a node (say n) by clicking it, pick a predicate (say $p1$), pick a value (say* True*), and click on "Set value". This would set the value for the the pair of selected node and predicate to* True*. In our heap pattern formalism from Defn. 5, this would amount to setting* $\mathtt{P}(n, p1) = \mathsf{True}$.

Similarly, picking a heap variable (say x) and setting a value (say Maybe*) for it using the form would amount to setting* $\mathtt{V}(n, x) = \mathsf{Maybe}$ *for the selected node n.*

Who are the Users?

Our system involves human users, so it is important to describe suitable candidates who will be able to use our system. We believe that a basic understanding of graphs or transition systems should be sufficient, and in fact very helpful in fully grasping what we expect from the user. This means that someone with an undergraduate degree in Computer Science or Mathematics, or indeed anyone who has learned about graphs and state transition systems would be a suitable user. A background in formal methods or verification is not required.

We now dive deeper into all the features provided by the interface, and the user can easily provide a graph pattern matching the formalism described in Defn. 5. One of the important goals of

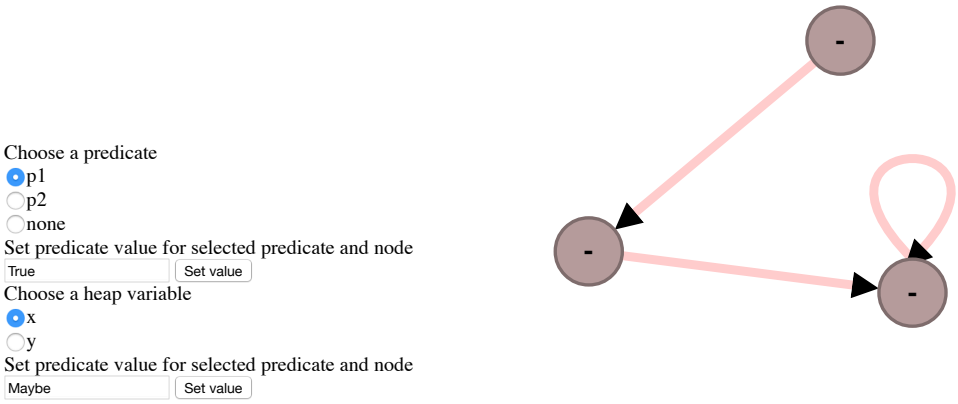

Figure 6.1: A simple interface to allow the user to graphically draw the heap pattern, and pick values for the predicate and heap variable labeling. The predicates and heap variables have been chosen and populated from a prior analysis. Nodes are represented by circles, and edges as arrows between nodes.

this design is that the user should not have to understand or deal with the formalism itself, but just figures, so that someone with no knowledge of software verification or heap modeling can perform the function of an Oracle.

A heap pattern is a labeled graph represented by the tuple $(N, \mathtt{V}, \mathtt{P}, E, \sigma)$, respectively containing the set of nodes, heap variable labeling, predicate labeling, edges, and summary function. Our interface allows the user to modify the values of each of these attributes of the pattern. We now describe each of these in greater detail.

Modifying Nodes

Interacting with nodes allows the user to modify the existing set of nodes. The following actions are available:

- Click anywhere on empty space to create a new node.
- Click on an existing node to select (or unselect) it. Selected nodes appear a lighter shade than

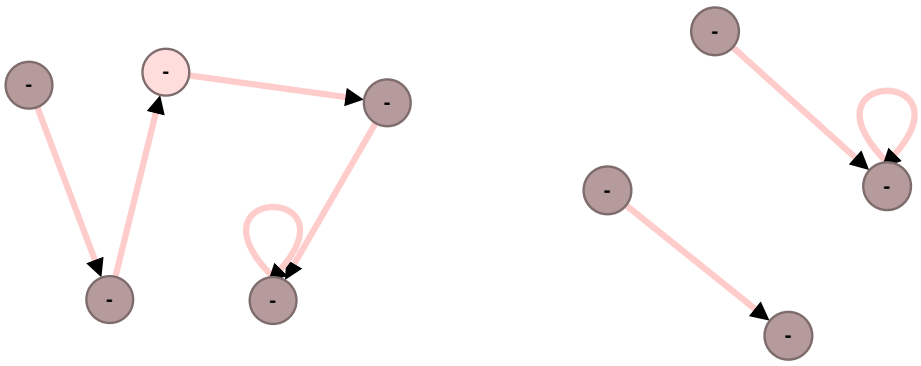

Figure 6.2: On the left, the original pattern graph has five nodes, with the middle node selected (indicated by the different color). Pressing the backspace key deletes the node, resulting in the graph on the right.

unselected nodes. Only up to one node can be selected at a time.

- Press the backspace key after selecting a node to delete it. This deletes all edges and resets other attributes relevant to the node.

Figure 6.2 illustrates the above points clearly.

Modifying the Heap Variable Labeling

We briefly described the mechanism for updating the heap variable labeling in Ex. 9. The interface itself is simple, as shown in Figure 6.1. In addition, there are some features that make it simpler to keep track of assignments, while preventing user mistakes.

The heap variable labeling is a map $\mathtt{V} : N \times \mathtt{Vars}_H \to \mathbb{B}_3$, meaning that each pair of node and heap variable must have a value. To indicate the current assignment, we label the node accordingly. For instance, if the current heap variables are x, y, z, and for node n, the values of $\mathtt{V}$ are $\mathtt{V}(n, x) = \mathsf{Maybe}, \mathtt{V}(n, y) = \mathsf{True}, \mathtt{V}(n, z) = \mathsf{False}$, then the node will get labeled as $x?y$. The $?$ indicates a Maybe value, the absence of a $?$ indicates True, and absence of the variable altogether indicates False. This keeps the labeling simple, preventing clutter from a large number of False

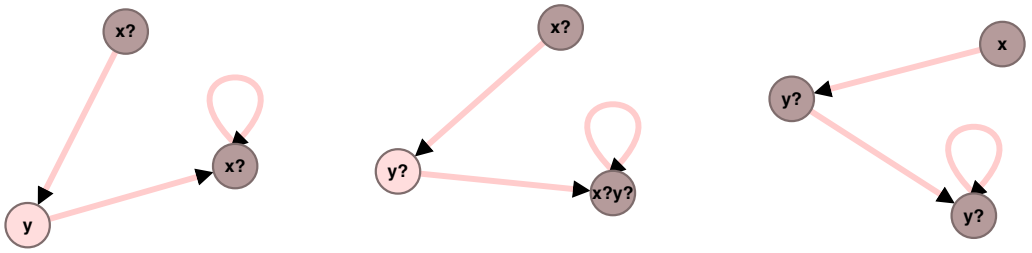

Figure 6.3: The first pattern on the left shows an existing heap variable labeling. We then set the value for y and the node with the self-loop to Maybe, which automatically sets the value for y and the selected node to Maybe, even though it was True earlier. Furthermore, in the second pattern, when x is set to True for the starting node, the value of x is unset for the node with the self-loop.

values, while still making it simple to get a quick idea of the current assignments. Finally, a node with no variable assignments will simply have the label $-$.

We note that `V` is not allowed to have arbitrary assignments. For instance, a variable x cannot be True for more than one node at the same time. Our interface takes care of such constraints as follows:

- Setting a variable to True for a node sets it to False for all other nodes automatically.
- Setting a variable to Maybe for a node sets it to Maybe for the node where it might currently be True.

Figure 6.3 further illustrates how heap variable labeling works.

Modifying the Predicate Variable Labeling

We briefly described the mechanism for updating the predicate variable labeling in Ex. 9. The interface itself is simple, as shown in Figure 6.1. In addition, there are some features that make it simpler to keep track of assignments.

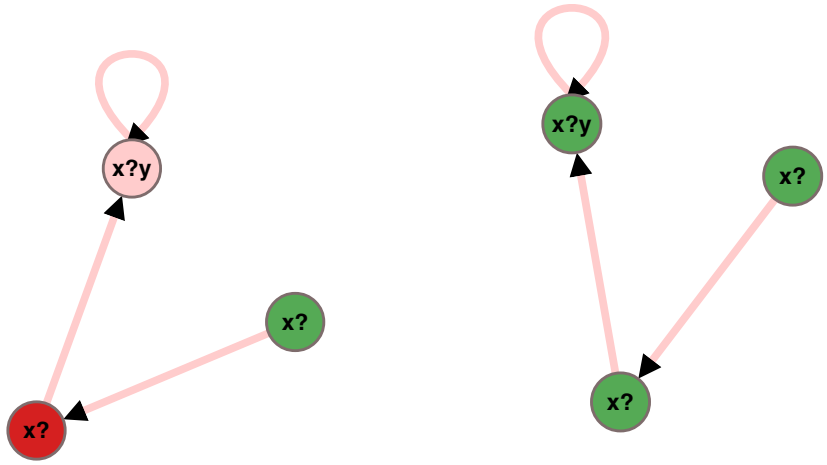

Figure 6.4: On the left, predicate $p1$ is selected, and it has values True, False, and Maybe in order of the arrows, indicated by the appropriate colors. Then we switch the "live" predicate to $p2$, which happens to be True for all nodes, so the color switches to green for each node.

The predicate variable labeling is a map $\texttt{P} : N \times \texttt{Vars}_P \to \mathbb{B}_3$, meaning that each pair of node and predicate variable must have a value. To indicate the current assignment, we use color coding for the nodes - $\texttt{P}(n, p) = \text{True}$ imples green, $\texttt{P}(n, p) = \text{False}$ imples red, and $\texttt{P}(n, p) = \text{Maybe}$ implies light pink. Notice that the pattern graph can have multiple predicates available at a time, so the color coding depends on a predicate that is "live" at a given time. Predicates can be made live by simply selecting them in the interface shown in Figure 6.1.

Figure 6.4 illustrates how predicate variable labeling looks.

Modifying Edges

Edges are represented by the map $E : N \times \texttt{Fields} \times N \to \mathbb{B}_3$. For our case, since we're only dealing with a single field in the interface, the map can be simplified to $E : N \times N \to \mathbb{B}_3$, meaning that each pair of nodes have an edge between them in either direction. For two nodes m, n, $E(m, n) = \text{True}$ implies that a green edge exists pointing from m to n. $E(m, n) = \text{Maybe}$ implies that the edge is light pink instead. If $E(m, n) = \text{False}$, then this is indicated by the absence of an edge. Once again, this is to make sure that the pattern graph isn't overly cluttered because of too

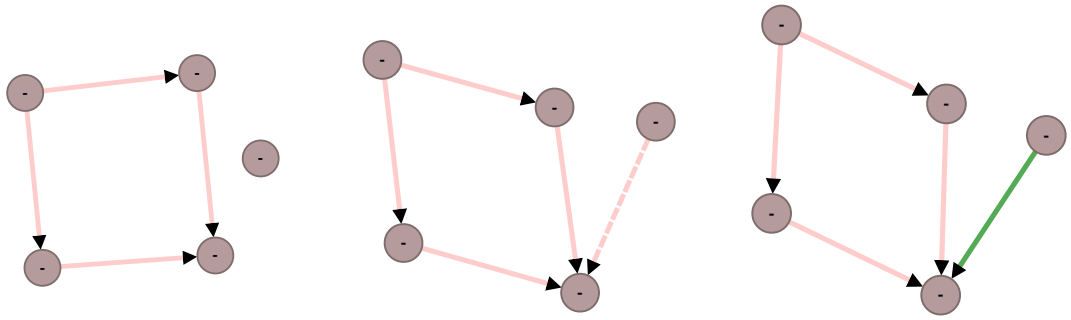

Figure 6.5: The first pattern on the left has a node n that is not connected to any other node. We drag from n to another node to create a new edge, which is also selected in the second pattern (indicated by the dotted line). For the selected edge, we press the M key, which turns it into a True (green) edge, in the third pattern on the right.

many False edges. An edge is also a link for the force-directed graph in D3.

Interacting with edges is fairly simple:

- Drag the mouse from a node to another to create an edge between them, in the direction of dragging.
- Click on an existing edge to select (or unselect) it. Selected edges appear dotted, as opposed to solid for unselected edges. Only up to one edge can be selected at a time.
- Press the backspace key after selecting a edge to delete it.
- Pressing the M key for a selected edge toggles between True and Maybe values for that edge (respectively green and light pink).
- Pressing the R key for a selected node creates a self-loop on that node. Each node can have only one self-loop, and all edge interactions work for it just like other edges.

Edges are normally straight, but bidirectional edges are rounded for better visibility. Figure 6.5 illustrates how the interface allows for interaction with edges.

Just like `V`, E is not allowed to have arbitrary assignments. For instance, a True edge going out

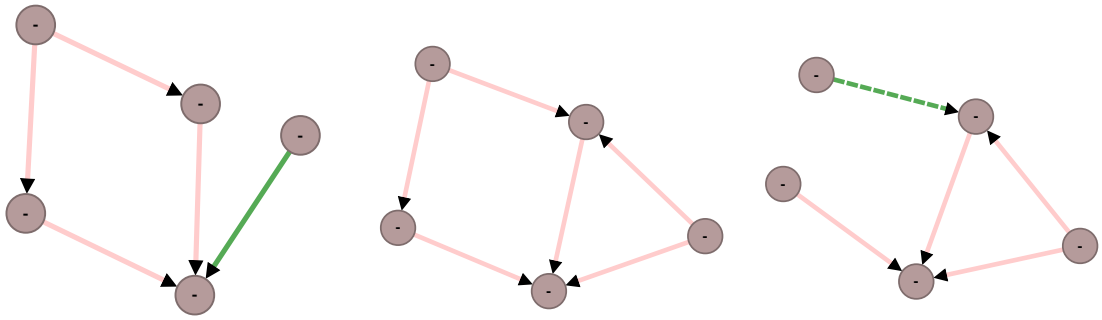

Figure 6.6: The first figure on the left has a node n with a True outgoing edge. We create another outgoing edge from n, which is a Maybe edge by default, but it also turns the True edge to Maybe. In the third pattern on the right, we turn a Maybe edge to True, which results in the other Maybe edge going out of it disappearing (becoming False).

from a node implies that there cannot be any other edge going out of the same node. Our interface takes care of such constraints as follows:

- Setting an edge (from node m to n) to True sets all outgoing edges from m False automatically i.e. deletes them.
- Setting an edge (from node m to n) to Maybe sets any True outgoing edge from m to Maybe automatically.

Figure 6.6 illustrates how constraints on E are enforced by the interface.

Modifying the Summary Function

The summary function $\sigma : N \to \mathbb{B}$ indicates where a node is a summary node. Our interface indicates summary nodes by making them larger than other nodes. Selecting a node and pressing the S key toggles the value of σ for that node. Figure 6.7 illustrates an example of this.

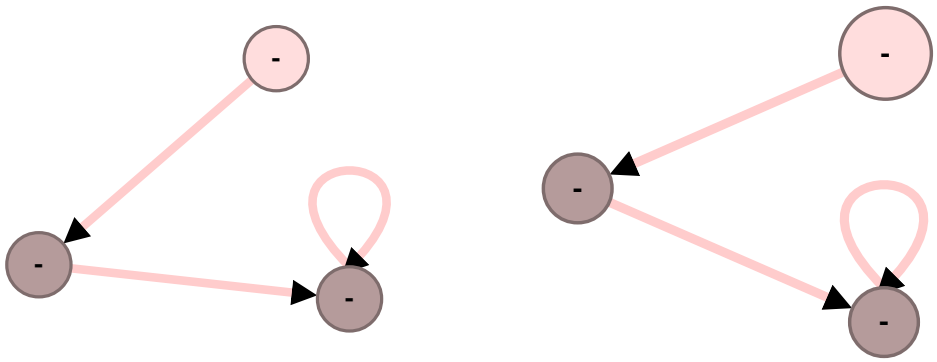

Figure 6.7: The pattern on the left has three nodes, one of which is selected. On pressing the S key, it becomes a summary node, indicated by the larger size in the pattern on the right.

6.3 User Interaction

The generated patterns are used by the PROVEIT algorithm as candidates for annotating vertices in the program unwinding tree. The following steps describe how interaction with the user works. We will later elaborate the same with an example.

1. At any point, interaction with the Oracle (human user) is limited to requested patterns for a single vertex of the program unwinding tree.
2. Start with $H^+ = H^- = \{\}$. At this point, the user can provide an arbitrary pattern to begin with.
3. If the pattern (we call it C) is such that $\forall h \in H^+, h \preceq C$ and $\forall h \in H^-, h \not\preceq C$, then C is sent to the verifier. On the other hand, if there is any heap in the existing sets H^+, H^- for which these conditions are not satisfied, it is highlighted to the user.
4. If the pattern is accepted in Algorithm 9, then no more input is requested from the user.
5. If the pattern is not accepted, one or both of H^+ or H^- are updated, and the user needs to update the provided pattern. Return to Step 3.

As described in Defn. 6, each individual heap can also be presented as a pattern that represents exactly that heap. As a result, all the individual heaps in H^+ and H^- are also patterns, but with limited properties and and no Maybe assignments to variables and edges, and no summary nodes. We now present an example to illustrate how the above steps work.

6.3.1 Illustrating User Interaction

We note that all heaps have a special `NIL` node, which represents the `NULL` address. `NIL` nodes cannot be summary nodes, and have no outgoing edges. In this example, we consider the program `alt_list_simplified` in Figure 6.8 that creates a linked list with alternating True and False values for the predicate p.

The user is not expected to know what the program or the verification algorithm does, and will only be presented with examples. User insight is expected based solely on those. We assume that only a single predicate $p \in \texttt{Vars}_P$ exists, so all node colors only refer to that single predicate. We start with $H^+ = H^- = \{\}$, at which point the user can't really provide any insight. It might be worthwhile to have a default candidate be provided by the interface itself at this point, but we still leave it to the user just in case they have additional insight about the program from other sources. Suppose the user just provides a pattern with a single node labeled `NIL` (Figure 6.9).

At this point, we get a new concrete heap back from the verifier (Figure 6.9). The user once again tries to respond in a simple way by providing the heap returned by the verifier as a pattern (Figure 6.9). In response to the pattern provided by the user, the verifier returns a new heap as part of H^+, shown in Figure 6.10.

Now is the first time the user needs to think of a non-trivial pattern, because the two heaps we have in H^+ are very different. At this point, the user makes two attempts, a wrong one and a right one, and we describe what happens in each case in Figure 6.11. The right pattern uses a summary node, because summary nodes can abstract out zero or more concrete nodes. Similarly, Maybe edges are used to indicate that the edge may or may not actually exist in a concrete pattern represented by the heap pattern.

```
1  void alt_list_simplified() {
2    // set of predicates: {p}
3    // allocate an initial node with predicate value TRUE
4    var x = alloc();
5    bool p_val = TRUE
6    set(x, p) = p_val
7    // set y to be the tail of the list
8    var y = x;
9    y->next = NULL
10   // LOOP-CONS: build a list with alternating Boolean values.
11   while (non_det()) {
12     // allocate a new node with alternating value
13     var z = alloc()
14     p_val = !p_val
15     set(z, p) = p_val
16     // append it to the existing list
17     y->next = z
18     y = z
19   }
20   checkpoint:
21   // rest of the program
22   return;
23 }
```

Figure 6.8: `alt_list_simplified`: a slightly modified and simplified version of the SV-COMP benchmark program `alt_list` (Figure 2.1) that constructs a list with cells that store alternating Boolean values for a known predicate. We will demonstrate concrete heaps and patterns for the prgram location at the label `checkpoint`.

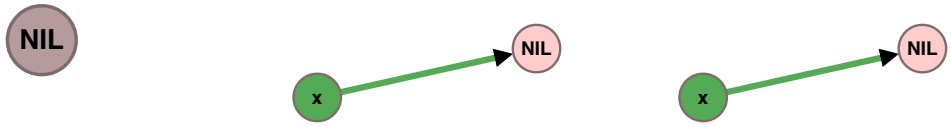

Figure 6.9: On the left is the initial pattern candidate provided by the user. At this point, the user does not know what the program is doing, or what program location the candidate is being requested for. As a response, the verifier returns the concrete heap in the center as a positive example, indicating that the pattern provided by the user should be able to cover it. The user in response eagerly provides the same positive example as a candidate pattern.

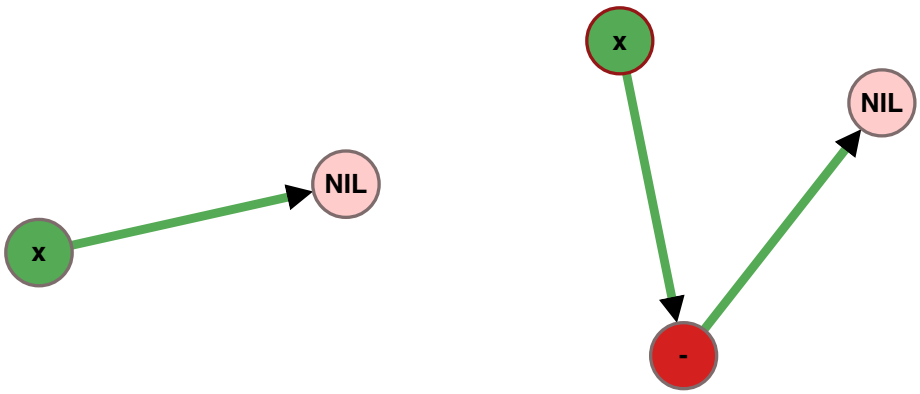

Figure 6.10: The set H^+ after the user has provided two candidate patterns to the verifier.

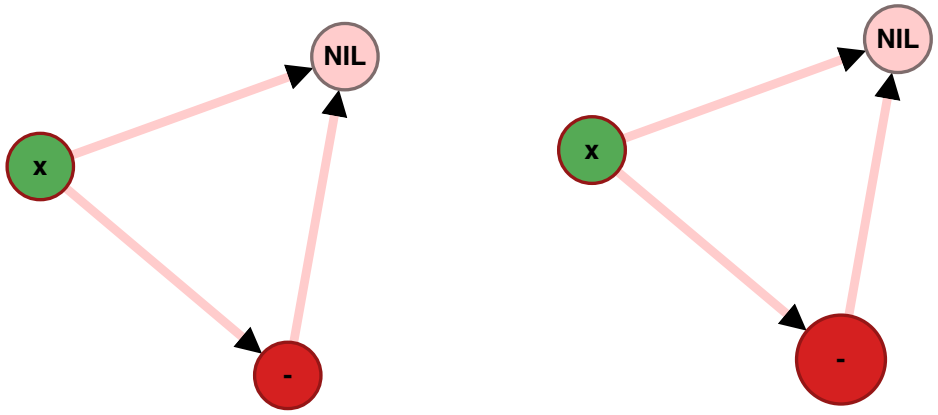

Figure 6.11: The user first provides the pattern on the left as a candidate. But we note that this pattern does not actually cover the first heap in H^+ in Figure 6.10, so the interface highlights the first heap, and the user has to correct their input. After making one of the nodes a summary node (indicated by the larger size in the pattern on the right), the pattern starts to be entailed by both our heaps in H^+, and the interface passes it on to the verifier.

After several back and forth interactions with the verifier, we might end up in a state where we have several positive examples, as shown in in Figure 6.12. The user now begins to get an idea of what the program might actually be doing at this program location. It looks like the program constructs "alternating" linked lists that always start with a node where the predicate is True. The heap variable x acts as the head of the list.

After seeing all these examples, the user submits the pattern in Figure 6.13, hoping that it would be accepted by the verifier. It might well be, depending on what state the verifier is in. But in our example, the pattern provided by the user turns out to be too general, and the verifier returns a negative example for the first time. Finally, H^- is non-empty, as looks as shown in Figure 6.14. Picking up on this, the user makes a correction, submitting the pattern in Figure 6.15, which is finally accepted by the verifier.

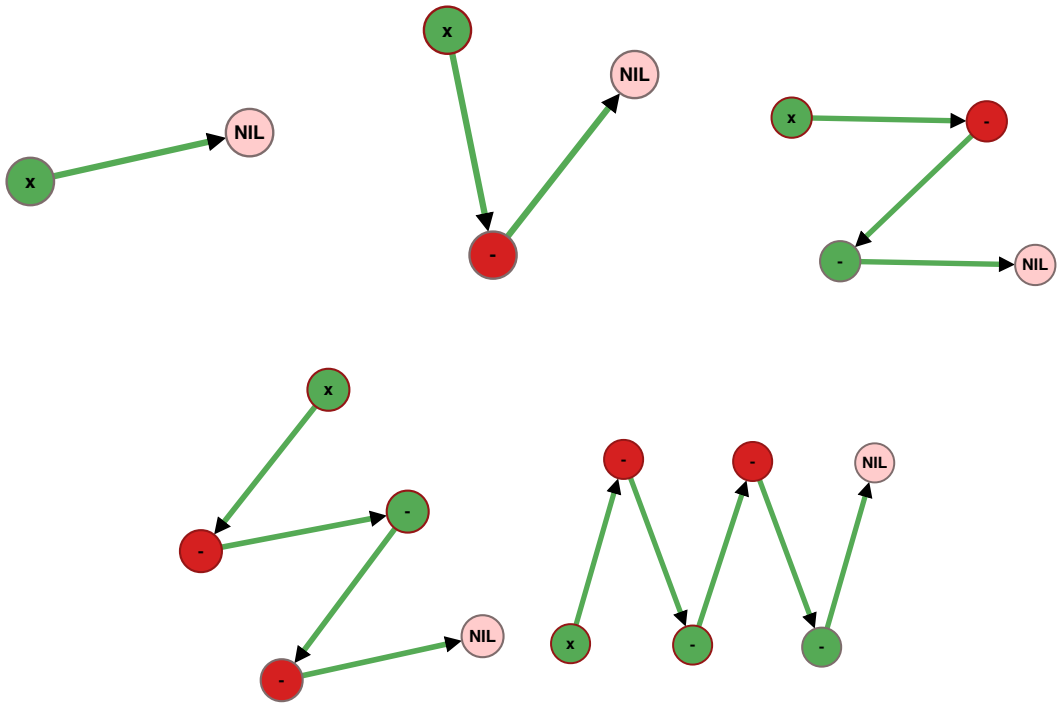

Figure 6.12: The set H^+ of positive examples, after a few back and forth interactions between the user and the verifier. Note that a suitable candidate has still not been found, and H^- is still empty.

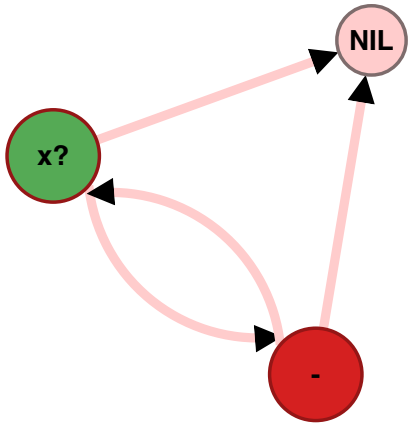

Figure 6.13: This candidate pattern is almost right, it captures the alternating list property, but has a problem that results in a negative example, shown in Figure 6.14.

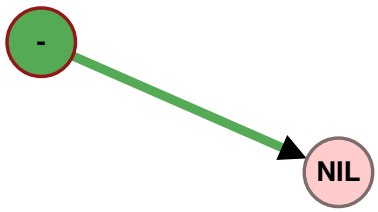

Figure 6.14: The set H^-, with a simple heap that indicates that the first node does not have x pointing to it. This heap is allowed by the pattern in Figure 6.13, but not by the pattern in Figure 6.15.

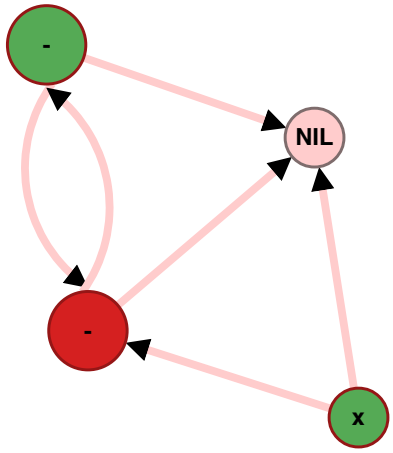

Figure 6.15: The final pattern that is accepted by the verifier. Note that this is not the strongest pattern indicating exactly the right heaps. PROVEIT does not need the strongest pattern, but the one that can be part of the interpolant. Too strong a pattern can lead to a lot of the vertices in the unwinding tree ending up uncovered. Too weak might mean the interpolant breaks down.

6.4 Analyzing our Interface

Our example in §6.3.1 gives one possible user interaction flow for the `alt_list_simplified` program. In this section, we present a broader discussion of the capabilities and limitations of our interface. This is important to understand what kind of programs can be handled by the heap pattern language and PROVEIT. They key goal of PROVEIT is to make it easier to provide a way to input and use unbounded heap patterns.

The visual graphical representation of heap patterns in our interface is based on Defn. 5. While the formalism is more difficult for humans to reason about, graphs are easier to understand. Nonetheless, the utility of our algorithm is limited by two things - (1) the expressiveness of the heap pattern language, and (2) the complexity of patterns that humans can provide. We now discuss these points in more detail.

6.4.1 Expressiveness of Heap Patterns

In Figure 6.15, we saw a heap pattern representing an alternating list. While the pattern may not be able to perfectly capture all kinds of alternating lists, it was sufficient for the example at hand. Similarly, a pattern might also over-generalize, but it might work for a certain node in our unwinding tree. This is a great feature of interpolant-based techniques, where as long as our interpolant is able to label an error node unreachable, it is sufficient and nodes don't need to be labeled with the strongest possible labels. Whether or not a pattern is sufficient depends significantly on the requirements of the underlying verification algorithm and the property to be proved. In this section, we demonstrate some examples of heap patterns that can represent some interesting shape properties. Notice that our current implementation only supports nodes that have a single pointer field. In the future, we would like to support nodes with multiple pointer fields, which would allow us to deal with structures such as doubly linked lists, trees, and even other shapes that are less common.

We now demonstrate some interesting patten graphs, some of which are related to examples featured in the SV-COMP [sv-15] benchmarks.

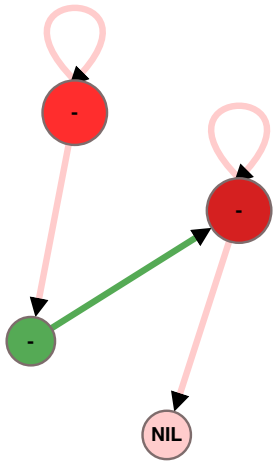

Figure 6.16: An example heap pattern that represents singly-linked lists that have one True node for a fixed predicate p. This node is always reachable from the head of the list.

Reachability

Consider a program that constructs a singly-linked list, such that for a fixed predicate p, there exists a single node in the list where p is True. Essentially, this node is always reachable from the head of the list, regardless of how many nodes the list has. An example of a heap pattern that could capture such a property is shown in Figure 6.16.

Conditional Predicates

In this example, we consider two predicates p_1 and p_2. Suppose a program constructs a singly-linked list such that for each node in the list, p_1 and p_2 always get opposite values. That is, if p_1 is True, then p_2 must be False, and vice versa. Figure 6.17 demonstrates a heap pattern which captures this property. The figure on the left is what the pattern looks like when p_1 is selected in our interface, and the figure on the right is when p_2 is selected. $x \in \texttt{Vars}_H$ is the head pointer for the list.

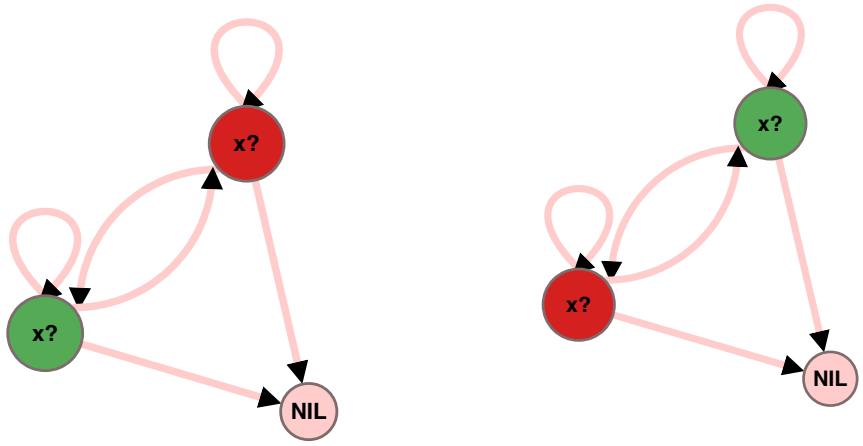

Figure 6.17: A heap pattern which demonstrates a list for which two predicates p_1 and p_2 cannot have the same value for any node.

Two-part List

Consider a program that for a fixed predicate p constructs a singly-linked list such that the list is composed of two distinct parts. For the first part, p is True and for the second part, p is False. A pattern that represents this is shown in Figure 6.18.

The last few examples show that useful properties related to heap data structures can be expressed using our framework. In the future, we would like to explore whether our heap pattern formalism has an equivalent in logic for describing heaps. At the same time, we note that PROVEIT itself is not strictly tied down to a single formalism, and it is possible to annotate nodes in the unwinding tree with other kinds of structures or formulas, and provide an Oracle that can work with them.

6.4.2 Understandability of the Interface

While our interface has not been tested at a large scale, we let six different users try it out. All of them had a computer science/electrical engineering undergraduate degree with programming experience. Three users were graduate students in computer science, while three others were software engineers.

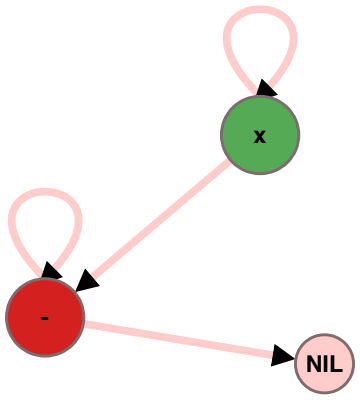

Figure 6.18: A heap pattern that represents a list made up of a True segment followed by a False segment.

We showed them two different examples. The first one was the alternating list example illustrated in §6.3.1. Three out of six users settled on the pattern shown in Figure 6.15. Two users provided a less abstract pattern, which still satisfied our conditions based on the presented examples, while one user struggled to find a pattern at first, being confused by the definition of summary nodes. The second example we presented after the previous one was the simple reachability example described in §6.4.1, and all users were able to describe the pattern we had in mind. While our interface obviously needs a lot more testing and results at a larger scale, the initial experience was encouraging, and we received constructive feedback that would help improve the interface, and possibly open it up to more users, perhaps with lesser expertise.

Feedback about the Interface

Based on the experience of our small set of users in §6.4.2, we received interesting feedback that would make sense to address as part of future work. The key points are listed here

1. As seen in Figure 6.12 and also noted earlier in §6.2.1, force-directed graph layouts aren't fully static, and can change even when only local alterations are made to the graph. While an aesthetic automated layout is desirable, some users complained about the lack of consistency

in the rendering, leading to difficulties in fully understanding all examples. We need to fix this by using alternative layout algorithms, or modifying our force-directed layout to render in a more predictable manner.

2. One user found it hard to fully grasp the idea of summary nodes. While it was clear that summary nodes need to be able to represent multiple concrete nodes in individual heaps, it wasn't entirely easy to visualize this.

3. Some users pointed out that for programs with larger number of predicates, our scheme of coloring nodes based on selected predicate and its value could become difficult to understand. This is true, and so far all our examples have worked out with up to two or three predicates, but we'd like to think of a more natural way of representing predicate values in the future.

4. Everyone felt that it would be really helpful to have live feedback on the pattern being built by the user. Right now, the user constructs a heap pattern and submits it, and then gets to know of a positive example not matched by it, or a negative example matched by it. For more complex programs, it would be useful to have a measure of "progress" that would allow users to make a more directed effort towards providing the right pattern, rather than just having to make a guess. This is a challenging problem, and worth thinking about. This is even before the pattern is even submitted to the verifier, which might add more examples.

6.4.3 Overall Implementation

We now briefly describe our first-cut implementation of PROVEIT. It is meant to be a simple proof of concept. Here are some of its features.

1. The three key steps of PROVEIT are EXPANDP (Algorithm 4), REFINEP (Algorithm 5), and COVERP (Algorithm 6). We built these on top of the Impact algorithm in CPACHECKER, which is a great tool for configurable software verification. CPACHECKER is a generic framework that can support a lot of verification algorithms [BK09], and has a good implementation of the Impact algorithm from [McM06]. We were able to extend it by implementing the heap and heap pattern formalism in CPACHECKER, and annotating nodes with heap patterns instead of predicates.

2. We implemented the Post operator (Defn. 9) for heap patterns, which allows us to generate new candidates in NEWCANDIDATE (Algorithm 9).

3. The first version of PROVEIT does not fully implement INTERPLEARNER (Algorithm 7) and NEWCANDIDATE (Algorithm 9). In our limited tests with users, we simulated those procedures manually.

4. Separate from CPACHECKER, we implemented our Oracle interface using D3. The two are supposed to interface using a common graph representation format, although this does not currently exist in our implementation.

Summary

In this chapter, we presented the web interface for the Oracle in PROVEIT- a human user. We described the interface design, and an overview of how a user would interact with the interface to provide a useful heap pattern that can be used in verification. We also discussed our overall implementation and its limitations. The next chapter presents some related work, and concludes with our contributions and ideas for potential directions to take.

Chapter 7

Conclusion

In this chapter, we briefly describe existing verification approaches, how they differ from our oracle-guided approach, and summarize the contributions in this thesis.

7.1 Related Work

Counterexample Guided Abstraction Refinement A broad class of verifiers of programs and transition systems have been proposed that implement *counterexample-example guided abstraction refinement (CEGAR)* [CGJ+03]. The common structure of all of these analyses is that they maintain an approximate model of the possible runs of a system, and refine the model until it represents a proof of correctness by iteratively (1) choosing a path of execution p allowed by the model that, if feasible, constitutes a property violation, (2) refuting the feasibility of p, and (3) using the refutation to refine the paths of execution allowed by the model. The CEGAR-based analysis that is most closely related to the one proposed in this work is actually a *theoretical* analysis that chooses program facts from which to construct a refutation by querying a *widening Oracle* [BPR02]. The key property of the counterexample-guided analysis is that if there is sequence of widenings that the Oracle can possible choose to cause the analysis to verify a program, then the analysis will eventually verify the program successfully. Because the Oracle does not solve a distinct problem, but instead provides values to the analysis, it can be viewed as an agent of *angelic non-determinism* [BCG+10].

While PROVEIT also queries an Oracle, the Oracle solves a problem distinct from providing values to the analysis, namely an active learning problem over both positive *and negative* example heap graphs.

Predicate Abstraction Predicate abstraction is an abstract interpretation technique in which the abstract domain is constructed from a given set of predicates over program variables. The concrete states of a system are mapped to abstract states according to their evaluation under a finite set of predicates. Automatic predicate abstraction algorithms have been designed and implemented before for finite and for infinite state systems. Predicate abstraction is well established in the literature [BMMR01, HJMS02, HJMM04]. The primary limitation with most of these techniques is that predicates in logic cannot describe shapes. PROVEIT uses predicates to annotate heap patterns, relying on other technique to infer these predicates beforehand. Predicate abstraction is useful to do that, although we need more investigation into how to use it to infer the right predicates for our technique in a directed way and efficiently.

Interpolation Interpolants have been widely studied and used in model checking and software verification. In various contexts, interpolation can be used as a substitute for image computation, which involves quantifier elimination and is thus computationally expensive. The idea is to replace the image with a weaker approximation that is still strong enough to prove some property, helping to construct and inductive invariant. Interpolant based techniques typically examine symbolic executions (finite paths) through the program, explicitly enumerating paths and employing heuristics to avoid path explosion [AGC12, HHP10, McM06, RHK13]. Some of them use other optimizations to cover wider search spaces, or compute invariants more efficiently. PROVEIT also uses the notion of interpolants applied to sequences, but specifically for heap patterns. Existing techniques don't work for this case because they deal with formulas that cannot desribe heaps or shapes in memory.

Active Learning and Inductive Synthesis Active learning has been explored as yet another technique, which is particularly useful for dealing with verification of data structures. The framework proposed in [GLMN13] can model quantified invariants over linear data structures, and build poly-time active learning algorithms for them, where the learner is allowed to query the teacher with

membership and equivalence queries. The work in [Li14] has the overarching theme of specification mining - the process of inferring likely specifications by observing a design's behaviors. It includes CrowdMine [LSJ12], which is a game devised for finding patterns from system traces that can suggest likely specifications, and a discussion of the feasibility of converting natural language specifications to formal specifications. [PSM16] extends the data-driven paradigm for precondition inference, by showing how to iteratively learn useful features on demand as part of the precondition inference process, thereby eliminating the problem of feature selection that affects existing data-driven techniques.

A theory of formal synthesis in inductive learning is presented in [JS15]. While our work in this thesis focuses on verification, synthesis is the dual problem of finding programs from specifications, and in the case of inductive synthesis, by using examples. This paper formalizes oracle-guided inductive synthesis (OGIS), as a framework that captures a family of synthesizers that operate by iteratively querying an oracle. Counterexample-guided inductive synthesis (CEGIS) is then presented as an instance of OGIS.

The work in [PMP+16] describes a system called Ivy, whose key principle is that whenever verification fails, it graphically displays a concrete counterexample to induction. The user then interactively guides generalization from this counterexample. This process continues until an inductive invariant is found. Ivy searches for universally quantified invariants, and uses a restricted modeling language which ensures that all verification conditions can be checked algorithmically. All user interactions are performed using graphical models, easing the user's task.

Shape Analysis In program analysis, a shape analysis is a static code analysis technique that discovers and verifies properties of linked, dynamically allocated data structures in (usually imperative) computer programs. It has been applied to a variety of problems, including memory safety and checking state properties. For example, proving that two data structures cannot access the same piece of memory, or discriminating between cyclic and acyclic lists. Separation logic [CDOY11, Rey02] is one component of existing work on shape analysis. It extends the simple imperative programming language with commands (not expressions) for accessing and modifying shared structures, and for explicit allocation and deallocation of storage. Assertions are extended by introducing a "separating conjunction" that asserts that its subformulas hold for disjoint parts of

the heap, and a closely related "separating implication". Coupled with the inductive definition of predicates on abstract data structures, this extension permits the concise and flexible description of structures with controlled sharing. Separation logic is quite expressive, but the major challenge lies in finding a suitable decidable sub-logic that is expressive enough for a given domain.

In addition to the logic approach, memory graphs have been extensively explored for shape analysis. A parametric framework for shape analysis was presented in [SRW02], which can be instantiated in different ways to create shape-analysis algorithms that provide varying degrees of efficiency and precision. It also proposed three-valued logic structures, an idea we extensively use to model heap patterns in our work. Symbolic Memory Graphs (SMGs) [DPV13] are another effective approach, particularly for modeling extremely low-level operations. The heap patterns used in PROVEIT are partially inspired by SMGs, but work at a higher level that is more suitable for an external Oracle, and in an interpolation-based verification framework.

Crowdsourcing for Formal Methods The idea of using human input for assisting in computational tasks is not new. Human computation is a paradigm for utilizing human processing power to solve problems that computers cannot yet solve [VA05, QB11]. Formal verification techniques are currently computationally expensive/undecidable, or require highly specialized engineers with deep knowledge of software technology and mathematical theorem-proving techniques, making them expensive and time-consuming. CrowdMine [LSJ12] is a game devised for finding patterns from system traces that can suggest likely specifications. It transforms segments of a program trace into 2D images and then displays a small subset of them to a non-expert crowd in the form of a puzzle game, using the input to infer program specifications. A similar idea is present in [DDE+12], but the focus in on verifying security properties, which is achieved by having users play a game with ball-and-pipe puzzles.

The Crowd Sourced Formal Verification (CSFV) program by DARPA[1] is yet another major effort to investigate whether large numbers of non-experts can perform formal verification faster and more cost-effectively than conventional processes. The effort has produced some interesting games, which are listed under Verigames [ver14].

[1] Defense Advanced Research Projects Agency

Some other work includes [PMF13] which uses human input as test oracles for software testing, [LBG+13] which uses human domain-knowledge about a given SAT formula to obtain backdoor variables for the SAT formula, and [FLA+13] which is a system for creating, grading, and analyzing derivation assignments across arbitrary formal domains. Thus students act as users and the system checks their assignments (proofs) for correctness.

Crowdsourcing has been successfully used in other areas as well. Eyewire [eye12] is a popular example. Gameplay involves 3D puzzles and advances neuroscience by helping researchers discover how neurons connect to process visual information. Solving puzzles actually reconstructs 3D models of neurons. Eyewire requires no scientific background to play.

7.2 Summary

Our work can be summarized using the following key ideas:

1. We extended the interpolant-based verification algorithm from [McM06] to work for the domain of heap-manipulating programs.
2. We defined the heap pattern formalism for expressing sets of concrete heaps.
3. We introduced a framework for oracle-guided synthesis of heap patterns, which allows the verification algorithm to use an external Oracle for the generalization step.
4. We demonstrated one example of such an Oracle - a human user who is good at generalizing shapes, and can provide valuable insight to help find heap interpolants.

Our framework is very general, in the sense that it allows for any kind of "pattern" formalism to be used alongside a domain expert Oracle. This provides two advantages. Firstly, it makes it very easy to try a simple interpolation-based verification algorithm for a new domain, where one might not have good automated techniques for generalization, but good "intuitive" understanding about it. Secondly, it simplifies plugging in different Oracles into an existing verification algorithm, allowing for broader possible insight into the analysis. It is easy to extend this to the case of allowing for

multiple Oracles working side by side in a single analysis, providing complementary insight into a verification problem. In the future, a theoretical analysis of oracle-guided verification algorithms would be an interesting direction. Some of this has been approached in [JS15] as a theory of formal synthesis in inductive learning.

Bibliography

[AGC12] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Craig interpretation. In *SAS*, 2012.

[BCG+10] Rastislav Bodík, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. Programming with angelic nondeterminism. In *POPL*, 2010.

[BK09] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. *CoRR*, abs/0902.0019, 2009.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.

[BPR02] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS*, 2002.

[CDOY11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6), 2011.

[CGJ+03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5), 2003.

[Cra57] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(03):269–285, 1957.

[d3j16] D3 data-driven documents, 2016. https://d3js.org.

[DDE+12] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Nathaniel Mote, Brian Walker, Seth Cooper, Timothy Pavlik, and Zoran Popović. Verification games: Making verification fun. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 42–49, New York, NY, USA, 2012. ACM.

[DPV13] Kamil Dudka, Petr Peringer, and Tomás Vojnar. Byte-precise verification of low-level list manipulation. In *SAS*, 2013.

[Ead84] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:146–160, 1984.

[eye12] Eyewire, 2012. http://eyewire.org/explore.

[FLA+13] Ethan Fast, Colleen Lee, Alex Aiken, Michael S. Bernstein, Daphne Koller, and Eric Smith. Crowd-scale interactive formal reasoning and analytics. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 363–372, New York, NY, USA, 2013. ACM.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991.

[GLMN13] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. Learning universally quantified invariants of linear data structures. In *CAV*, 2013.

[HHP10] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. In *POPL*, 2010.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL*, 2004.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, 2002.

[JS15] S. Jha and S. A. Seshia. A Theory of Formal Synthesis via Inductive Learning. *ArXiv e-prints*, May 2015.

[LBG+13] Ronan LeBras, Richard Bernstein, Carla P Gomes, Bart Selman, and R Bruce Van Dover. Crowdsourcing backdoor identification for combinatorial optimization. In *IJCAI*, pages 3–9, 2013.

[Li14] Wenchao Li. *Specification Mining: New Formalisms, Algorithms and Applications*. PhD thesis, EECS Department, University of California, Berkeley, Mar 2014.

[LSJ12] Wenchao Li, Sanjit A. Seshia, and Somesh Jha. Crowdmine: Towards crowdsourced human-assisted verification. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1254–1255, New York, NY, USA, 2012. ACM.

[McM05] Kenneth L McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.

[PMF13] Fabrizio Pastore, Leonardo Mariani, and Gordon Fraser. Crowdoracles: Can the crowd solve the oracle problem? In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 342–351. IEEE, 2013.

[PMP+16] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 614–630, 2016.

[PSM16] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 42–56, New York, NY, USA, 2016. ACM.

[QB11] Alexander J. Quinn and Benjamin B. Bederson. Human computation: A survey and taxonomy of a growing field. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 1403–1412, New York, NY, USA, 2011. ACM.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[RHK13] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for Horn-clause verification. In *CAV*, 2013.

[SRW02] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 2002.

[sv-15] SV-COMP 2015 - 4th international competition on software verification, 2015. http://sv-comp.sosy-lab.org/2015/.

[VA05] Luis Von Ahn. *Human Computation*. PhD thesis, Pittsburgh, PA, USA, 2005. AAI3205378.

[ver14] Verigames, 2014. http://www.verigames.com/.

A Personal Note

Chronic depression hindered my ability to make progress with research, and the completion of this thesis. For several months, I suffered without respite. I also missed several deadlines to submit this thesis. At the end, while the department refused to make an exception to help me, it was the Disabled Students Program (DSP) and some people connected with it who truly understood what I was facing, and helped me. On the other hand, several conversations I had with people in the EECS department (none of whom I wish to name) were not only unhelpful, but some were also inconsiderate. I do not wish to blame individuals, but it was very clear how poorly equipped academic departments can be, when it comes to systematically helping students suffering from mental health issues.

If you're a graduate student who has been suffering from mental health issues, remember that help is available, but unfortunately you might have to reach out yourself. Systems are meant to help people, but they aren't perfect and often fail. I hope that briefly addressing this issue here will serve as encouragement to someone, and help shine some light on the issue of mental health in academia.