DRAFT: UNDER SUBMISSION

# Invariant Synthesis for Sound but Incomplete Verification Engines

D. Park D. Neider S. Saha, P. Garg P. Mahdian P. Madhusudan

University of Illinois at Urbana-Champaign

## Abstract

We present a new learning-based technique to synthesize conjunctive invariants for sound but incomplete verification oracles. Our learning framework encodes *nonprovability* information provided by the verification oracle as constraints on the set of conjuncts that form the invariant, and reduces this learning to ICE-learning algorithms for conjuncts. We build new ICE learning algorithms for conjuncts based on constraint-solving and machine-learning that learn small invariants. We apply our learning framework in the setting of a verification engine that uses natural proofs for heap verification, exhibiting how natural proofs can provide the required non-provability information. We implement our techniques and extensively evaluate them and show that we can, using a combination of learners, automatically synthesize small inductive invariants for a large suite of programs that dynamically manipulate the heap.

***Categories and Subject Descriptors*** CR-number [*subcategory*]: third-level

***Keywords*** keyword1, keyword2

## 1. Introduction

The paradigm of *deductive verification* [32, 44] combines manual annotations and semi-automated theorem proving to prove programs correct. Programmers annotate the code they develop with contracts and inductive invariants, and use high-level directives to an underlying mostly-automated logic engine to verify their programs correct. Several mature tools have emerged that support such verification, in particular tools based on the intermediate verification language BOOGIE [4] and the SMT solver Z3 [26], like VCC [19] and DAFNY [54]. Several applications that use such tools to prove systems correct using manual annotations have been developed, including Microsoft Hypervisor verification [20], reliable systems code like VERVE[73], ExpressOS [56], Ironclad apps [42], and distributed systems in IronFleet [43]. Fully automated use of such engines for shallow specifications have also emerged, such as CORRAL [52] for verifying device drivers, CST [15] to certify transactions in online services, and GPUVERIFY [7] to ensure race-freedom in GPU kernels.

Viewed from the lens of deductive verification, the primary challenges to automating verification are two-fold. First, even when strong annotations in terms of contracts and inductive invariants are given, checking whether the resulting verification conditions are valid is often undecidable (for instance, in reasoning about the heap). Second, in order to automate program verification, we need to be able to automatically synthesize loop invariants and strengthening of contracts to prove a program correct, lifting this burden that is currently borne by the programmer.

One approach to the first problem is to build sound but incomplete verification engines for validating verification conditions that work for most programs, and thus skirt the undecidability barrier. Several such techniques already exist; for instance, for reasoning with quantified formulas, tactics such as E-matching [25, 27], pattern-based quantifier instantiation [27], and model-based quantifier instantiation [37], are effective in practice, though they are not complete for most background theories. In the realm of heap verification, the so-called *natural proof* method explicitly aims to provide automated and sound but incomplete methods for checking validity of verification conditions with separation logic specifications [18, 55, 61, 65]. The methods proposed search for proofs based on induction on recursively defined data structures that unfold recursive definitions depending on how the program manipulates the heap and unify terms to prove the verification condition valid. Moreover, the search for such natural proofs is reduced to validity problems in *decidable logics with quantification* that enables an efficient search for such proofs using SMT solvers.

Turning to the problem of invariant generation, white-box techniques (such as interpolation [47, 57] and IC3 [11]) that generalize from information gathered in proving underapproximations of the program correct are quite effective [8], but their efficacy in dealing with programs where even verification is challenging is unclear. However, a class of *black-box* methods have emerged recently that propose to *learn* invariants from information provided through concrete program configurations. The key idea here is that the invariant synthesis engine is largely agnostic of the program that is being verified, but is data-driven, relying on a verification oracle that examines candidate invariants it proposes and refutes them with concrete program configurations. In particular, the ICE learning model [34] is a robust learning model for invariant synthesis that asks the verification oracle to provide positive counterexamples (if the invariant is too strict), negative counterexamples (if the invariant is too liberal leading to violating assertions), or implication counterexamples (if the invariant is non-inductive). Several effective ICE learning algorithms have been proposed, based on using constraint-solvers [34], stochastic search [69], and machine-learning algorithms for learning decision trees [36].

*The primary aim of this paper is to explore black-box learning-based invariant synthesis algorithms that can work with sound but incomplete verification oracles, using non-provability information that the oracle can provide in lieu of concrete counterexamples.*

The main advantage of black-box invariant synthesis (argued for example in [36]) is that the invariant generator, being agnostic of the program and the property being verified, is not plagued by the complexity of the logic expressing verification conditions, but instead just works on finding some formula that satisfies the constraints imposed by the finite set of concrete program configurations that the verification oracle has provided. This technique is hence especially attractive in the setting where the verification oracle needs to reason with complex undecidable theories. However, the learning paradigm assumes that the verification oracle can return *concrete program configurations* that refute candidate invariants. This is problematic when we use sound but incomplete procedures as verification oracles, as such oracles (including those based on natural proofs) typically cannot provide such concrete counterexamples. In fact, one criticism of the claim that black-box learning can help in synthesizing invariants for programs with complex properties is that complete verification oracles will anyway not exist for such programs.

The first technical contribution of this paper is a technique that allows the combination of sound but incomplete verification oracles with learning algorithms to synthesize invariants, under certain assumptions that the oracle can return some non-provability information when verification fails. In particular, we consider the problem of learning conjunctive invariants where conjuncts are composed of predicates drawn from a large class of enumerated predicates, and show that verification oracles that can return information regarding provability and non-provability of verification conditions involving these predicates can be used by a learner to synthesize invariants.

For conjunctive invariants, we show that non-provability information can be encoded using a particular class of constraints on the set of predicates that can appear conjunctively in the invariant. We introduce a learning model for learning sets under such constraints (called LSUC) that can be paired with the oracle to synthesize invariants. Furthermore, we show that we can reduce LSUC, the problem of learning sets under constraints, to the more well-studied problem of ICE-learning of conjunctive formulas [34].

We then turn to study the problem of ICE-learning conjuncts. The Houdini algorithm is in fact a classical ICE learner of conjunctions, and can be readily applied to learn invariants from nonprovability information [31]. However, Houdini is not property driven, and generates the (unique!) largest conjunctive inductive invariant for a program, which often is too large to be human-understandable. We hence turn to the problem of ICE-learning small conjunctions. The second contribution of the paper is the design of two new property-driven learning algorithms— the ICE-CS learner that uses constraint solvers to find the *smallest* conjunctive formula in each round, and the ICE-Haussler algorithm, an adaptation of an efficient machine-learning algorithm for positive/negative samples that has a bias towards small conjunctions (which not guaranteed to be minimal).

The third contribution of the paper is to apply the learning framework we develop for sound-but-incomplete verification oracles based on natural proofs for heap verification. Using the fact that natural proofs reduce (soundly but incompletely) validity tasks to validity of formulas in decidable logics that can be solved using SMT solvers, we show that non-provability information can be extracted from the *models* found by SMT solvers when failing to prove validity. Combined with our reduction to ICE-learning, this gives us a robust framework for learning invariants for heap verification using natural proofs.

Our final contribution is a thorough evaluation of our learning framework for natural proofs using the various learners. We implement our technique by using the natural proof engine VCDRYAD[61] (which extends VCC and uses Boogie/Z3 engines) and pair it with the Houdini, ICE-CS, and ICE-Haussler learners. We evaluate it over a large class of heap manipulating programs and show that our technique is extremely effective in automatically synthesizing loop invariants and strengthening contracts to prove these programs correct. Our technique searches for conjunctive invariants involving a very large class of predicates (hundreds), where many of them will not hold for any given program, and even a few executions on simple tests would refute them. Consequently, we also suggest a technique that uses dynamic executions to effect an initial pruning of candidate predicates,

and show that this results in significant speedup in verification.

Our experiments show a particular combination of the above techniques especially effective— to use testing-based pruning to prune predicates initially, follow this with a Houdini learning to find a property-agnostic invariant (which is often very large), and then use this set of predicates to further learn, using ICE-CS or ICE-Haussler algorithms, a small inductive invariant that is property-driven. This combination gives us a fairly powerful automatic verification tool that rivals the state-of-the-art in heap verification today.

**Related Work**

Our work builds upon DRYAD[61, 65], a dialect of separation logic [66], and the natural proof technique line of work for heap verification developed by Qiu et al. [55, 61, 65], which build sound but incomplete techniques for verification for fairly expressive user-defined recursive predicates. In [61], the authors present VCDRYAD, which extends the VCC [19] framework to provide an automated deductive framework against separation logic specifications for C programs based on natural proofs. In contrast to the work we present here, VCDRYAD assumes that all loop invariants as well as strong pre- and postconditions are given. A similar technique for automated reasoning about dynamically manipulated data structures has recently been proposed by Chu et al [18]. This method relies on the unfold-and-match paradigm which employs systematic transformation steps of folding/unfolding the recursive rules. [17] is an earlier work that also uses similar ideas for folding and unfolding recursive rules.

There is a rich study of decidable fragments of separation logic in the the context of deductive verification in literature. Smallfoot [5, 6] implements a decision procedure for a fragment of separation logic for pure spatial properties of list segments, and there have been significant extensions to it [22], with some incorporation of lists with arithmetic [59]. Encoding separation logic fragments in SMT to get decidability has been developed as well— by reducing separation logic over lists to a logic over graph reachability and stratified sets [62], and extending them to trees and data [63, 64]. The decision procedures for these logics, however, do not extend to arbitrary user-defined predicates (indeed, the problem is in general undecidable). More expressive fragments of separation logic have been shown to be decidable using monadic second-order logics on bounded tree-width graphs, but these are of high complexity [45]. Brotherston et al [12] show a symbolic heap fragment of separation logic to be decidable. Some of the above work can be adapted to give concrete counterexamples using which we can build ICE-learning algorithms. However, implementing black-box invariant learning using non-provability information for these approaches (even for complete procedures) using the framework proposed in this paper would be interesting future work.

One can broadly categorize algorithms for invariant generation into white-box and black-box techniques. Most popular white-box techniques include abstract interpretation [23], interpolation [47, 57], IC3 [11], as well as an array of techniques relying on constraint solving [21, 39, 40]. On the other hand, Daikon [29] and Houdini [31] are two popular black-box techniques for invariant generation, which both use learning of conjunctive Boolean formulas.

In fact, learning has recently regained interest in the area of verification, specifically as an effective means to generate loop invariants [16, 33–35, 50, 51, 69–72]. In this context, Garg et al. [34] proposed a general learning framework for invariants, called ICE learning, which extends the classical learning setting (i.e., learning from classified data points) with implications. Algorithms that operate in this framework have been proposed for learning invariants over octagonal domains [33, 36], universally quantified invariants over linear data structures [33, 35], Regular Model Checking [60], and a general framework for generating invariants based on randomized search [69]. In addition, the Houdini algorithm [31] can also be seen as an ICE learning algorithm, as we show in a later section. Finally, it is worth noting that learning from implications has independently also been investigated by Sharma et al. [72].

For the synthesis of invariants over dynamic data structures such as arrays and lists, several white-box techniques based-on Craig's interpolation [57] have emerged in recent years [3, 48, 58, 68]. For instance, Safari [3] is a model checker for (potentially universally quantified) safety properties of imperative programs over unbounded arrays. Abstract interpretation [23] has also been used for synthesizing invariants of array and heap manipulating programs [1, 24, 38]. Another example of such an approach is the framework by Bouajjani et al. [9, 10], in which the authors consider invariant synthesis for sequential programs manipulating singly-linked lists.

There has been a lot of work on synthesizing invariants for separation logic using shape analysis [13, 14, 28, 53, 67]. Recent work also includes synthesizing heap invariants by extending IC3 [49] and by discovering spatial interpolants [2]. Finally, there also exist several learning-based black-box techniques, including methods based on predicate abstraction [50] and a reduction to a special type of register automata [33, 35].

## 2. Black-box Synthesis of Inductive Invariants Against Sound but Incomplete Oracles

In this section, we outline the problem of learning invariants from a sound-but-incomplete verification oracle, the space of counterexamples that the verification oracle can return, and how these counterexamples model information about the non-provability of verification conditions.

Let us fix a program $P$ for the rest of this section that is annotated with assertions (and possibly with some partial annotations that describe pre-conditions/post-conditions). Let us also fix a sound-but-incomplete verification oracle SIVO.

The primary goal is to verify $P$ against the assertions in it using the verification oracle. Let us assume that the program contains an annotation-hole $A$ that needs to be filled in with some annotation in order to verify the program. We restrict the problem to synthesis of one annotation for simplicity; the framework we build smoothly extends to multiple annotations.

Our framework consists of pairing the verification oracle with a *learner* whose task is to synthesize the annotation $A$. In each round, the learner will come up with an annotation, and the verifier will attempt to verify the verification conditions that arise from plugging in this annotation. If it is unable to verify the program correct, then this could be either due to the fact that the annotation is incorrect (not an invariant), the annotation is inadequate to prove the assertions in the program, or the annotations are logically adequate but the verification oracle is unable to prove them. We would like the verification oracle to return information that guides the learner towards learning an annotation that is adequate to prove the program with the verification oracle.

When the learner proposes an annotation for $A$, and the verification oracle fails to prove the verification condition for a Hoare triple $\{\alpha\}s\{\beta\}$, either $\alpha$ or $\beta$ or both could involve the synthesized annotation. The verification oracle could simply just convey that the annotation for $A$ is insufficient to prove the program correct—however, this gives little information on what aspects of the annotation were problematic. With this little information, the learner would gain no information, and will essentially only be able to *enumerate* annotations in some manner, and hence will be inefficient.

In the following, we specialize the above learning-based synthesis to *conjunctive annotations* where formulas for annotations are a subset of conjuncts drawn from a fixed class of formulas. As we show, the verification oracle can more easily generalize the counterexamples to weaker or stronger formulas when examining conjunctions, and furthermore, we can build efficient learning algorithms (see Sections 4 and 5). Throughout this paper, we use the notation $\bigwedge S$ for a set of predicates $S$ as a shorthand for the conjunction $\bigwedge_{p\in S} p$.

We make a natural assumption for the verification oracle that it is *normal*, as defined below.

**Definition 1.** *A verification oracle is* normal *when for any finite set of predicates S, (a) the verification oracle can prove a Hoare triple $\{\alpha\}s\{\bigwedge S\}$ if and only if it can prove every Hoare triple $\{\alpha\}s\{p\}$ where $p \in S$, and (b) if the verification oracle cannot prove a Hoare triple $\{\bigwedge S\}s\{\beta\}$ then it cannot prove $\{\bigwedge X\}s\{\beta\}$ for any set $X \subseteq S$.*

### 2.1 Synthesizing Conjunctive Invariants and Encoding Non-provability Using Set Constraints

Let us fix a finite set of predicates $\mathcal{P}$. Our goal will be to synthesize formulas in the logic $\mathcal{L}$ that consists of conjunctions of predicates drawn from $\mathcal{P}$ (i.e., $\mathcal{L} = \{\bigwedge_{p\in X} p \mid X \subseteq \mathcal{P}\}$).

We show in this section that non-provability information of verification conditions by the verification oracle can be encoded using a set of *constraints* on the set of predicates used for annotations. This results in a learning setting, called *learning sets under constraints (*LSUC*)*, in which the task is to learn a conjunct that satisfy such constraints. We formulate this precise learning model in Section 2.2. In Section 3, we then reduce the LSUC problem to a more well-studied problem called *ICE learning* of conjunctions, and in Section 4 we present several ICE learning algorithms for conjunctions, which are hence solutions for LSUC.

***What information can a sound-but-incomplete verification oracle provide?*** The overall verification goal is to find an annotation for the annotation-hole $A$ that is a conjunct of a subset of predicates in $\mathcal{P}$ that leads the verification oracle to prove the program correct. For the following discussion, assume that the learner presents a candidate invariant $\gamma := \bigwedge S$ for some $S \subseteq \mathcal{P}$ to be used for $A$, but the sound-but-incomplete verifier SIVO is unable to verify the program using $\gamma$. Let us now consider each kind of verification condition and examine what useful non-provability information SIVO can provide the learner to help its future attempts to come up with an invariant. In the explanation below, $\alpha$ and $\beta$ are annotations, $s$ is a statement of $P$, and $\nvdash_{SIVO} \{\alpha\}s\{\beta\}$ denotes that the validity of the Hoare triple $\{\alpha\}s\{\beta\}$ is not provable by SIVO.

- If $\nvdash_{SIVO} \{\alpha\}s\{\gamma\}$ (i.e., we cannot prove a VC where the precondition is an existing annotation and the post-condition is the synthesized annotation $\gamma$), then it means that there is some predicate in $\gamma$ that we cannot prove to hold after executing $s$ from a state satisfying $\alpha$.
  Since $\gamma$ is the set of conjuncts of a set $S$, if the prover is unable to prove $\gamma$, then there exists some subset of $T_+ \subseteq S$ that the prover is not able to prove. Furthermore, clearly the prover cannot prove any set of conjuncts that includes some element of $T_+$.
  Hence, we can send the set $T_+$ to the learner and insist that in future rounds, the learner has to come up with a set $X$ such that $X \cap T_+ = \emptyset$. We call $T_+$ an *avoid set* sample.
- If $\nvdash_{SIVO} \{\gamma\}s\{\beta\}$ (i.e., we cannot prove a VC where the pre-condition is the synthesized annotation $\gamma$ and the postcondition is an existing annotation), then we know that the conjunction of predicates mentioned in $\gamma$ is not strong enough for the verifier to prove that $\beta$ holds after executing $s$. Now, let $T_-$ be a *superset* of predicates in $\gamma$ such that the verifier is unable to prove $\beta$ holds after executing $S$ with the conjunction of predicates in $T_-$ as a precondition. (We will show later how the verifier can come up with $T_-$ in some concrete verification scenarios.)
  We communicate $T_-$ to the learner and insist that in future rounds, the learner come up with a set $X$ such that $X \not\subseteq T_-$. This is correct since for any $X$ with $X \subseteq T_-$, the verifier will not be able to prove the verification condition with

$\bigwedge X$ as precondition since it is a weakening of $\bigwedge T_-$ (recall that SIVO is natural). We call $T_-$ a *differ set* sample.

- If $\not\vdash_{SIVO} \{\gamma\}s\{\gamma\}$ (i.e., we cannot prove the VC where both the pre- and post-condition is the synthesized annotation $\gamma$), then we cannot conclude that the annotation must be stronger or weaker than $\gamma$.

  We now assume that the verification oracle can come up with two sets of predicates $U_1$ and $U_2$ such that $U_1$ is a superset of predicates in $\gamma$ and $U_2$ is a subset of predicates in $\gamma$ such that the oracle cannot prove the verification condition $\{\bigwedge U_1\}s\{\bigwedge U_2\}$.

  Then we can communicate the pair $(U_1, U_2)$ to the learner and demand that the learner come up with a set $X$ such that *if* $X \subseteq U_1$ then $U_2 \cap X = \emptyset$. Intuitively, this condition is correct since if $X \subseteq U_1$ and $U_2 \cap X \neq \emptyset$, then the validity of the verification condition $\{\bigwedge X\}s\{\bigwedge X\}$ would imply the validity of $\{\gamma\}s\{\gamma\}$, which the verifier could not prove. We call the pair $(U_1, U_2)$ a *double constraint* sample. (Again, we will show later how such a double constraint sample can be found in the context of certain verification oracles such as natural proofs for heap verification.)

### 2.2 Learning Sets Under Constraints

The above analysis clearly shows that when the sound-but-incomplete verifier cannot prove the verification conditions resulting from plugging in a candidate annotation $\gamma$, it can return an avoid set $T_+$, a differ set $T_-$, or a double constraint $(U_1, U_2)$.[1] We can accumulate such constraints over successive rounds in a data structure called a *sample set*. A learning algorithm (annotation synthesizer) can then use this information to propose a new candidate annotation. The process of proposing candidate annotations and checking them repeats until the learning algorithm proposes a valid annotation or the learning algorithm detects that no valid annotation exists (recall that the set of predicates is assumed to be finite). This idea leads to two learning models: (a) a *passive* learning model that describes the learning in one iteration of the process described above, and (b) an *iterative* learning model that describes the learning as a whole.

For the sake of simplicity, let us assume that the learning algorithm accumulates the constraints returned by the verification oracle in a data structure that we call sample. Formally, a *sample* is a triple $\mathfrak{S} = (AS, DS, DC)$ where $AS \subseteq 2^{\mathcal{P}}$ is a set of avoid sets, $DS \subseteq 2^{\mathcal{P}}$ is a set of differ sets, and $DC \subseteq 2^{\mathcal{P}} \times 2^{\mathcal{P}}$ is a set of double constraints. We say that $X$ is consistent with a sample $\mathfrak{S} = (AS, DS, DC)$ if

- $X \cap T_+ = \emptyset$ for every $T_+ \in AS$;
- $X \not\subseteq T_-$ for every $T_- \in DS$; and
- if $X \subseteq U_1$, then $U_2 \cap X = \emptyset$ for every $(U_1, U_2) \in DC$.

[1] Existence of $T_+$, $T_-$, and $(U_1, U_2)$ is obvious. One can simply return the precise $\gamma$ that the learner gave. We will show later how the verifier can come up with non-trivial constraints.

***Passively learning sets under constraints:*** The *passive learning of sets under constraints (passive* LSUC*)* is a learning model where the learner needs to find, given a sample $\mathfrak{S} = (AS, DS, DC)$, some set $X$ consistent with it.

***Iteratively learning sets under constraints:*** In the *iterative learning of sets under constraints (iterative* LSUC*)*, the learner interacts with an oracle in order to learn *some* set in a target set $\mathcal{T} \subseteq 2^{\mathcal{P}}$. In each round, the learner has a sample $\mathfrak{S} = (AS, DS, DC)$, and learns a set $X$ consistent with the sample. If $X$ is in the target set $\mathcal{T}$, then the learner has learned the concept, and we stop. Otherwise, the oracle *adds* a new avoid set, a new differ set, or a new double constraint to the sample (consistent with targets in $\mathcal{T}$), and the process iterates.

Note that the iterative learning model is the one of primary interest. However, *any* passive learning model can be used in an iterative setting. Our goal is to build learners that (a) in each round, learn the set $X$ efficiently, and (b) minimize the number of rounds to learn a target concept. Any such iterative learner can be plugged in into our framework to obtain an *automatic* verification engine that synthesizes invariants that prove the program using the verification oracle.
We can in fact show a *relative completeness* result:

**Theorem 1.** *Let $P$ be a program with assertions and an annotation-hole and* SIVO *be a* normal *sound-but-incomplete verification oracle that can come up with some set constraint whenever an annotation fails to prove the program correct. Let $\mathcal{P}$ be a finite set of predicates, and assume that there exists some $S \subseteq \mathcal{P}$ such that the program verifies with the annotation $\bigwedge S$ under* SIVO*. Then, when the verification oracle is paired with any consistent* LSUC *learner (learner of sets under constraints), the learner will eventually learn an annotation that proves $P$ correct using* SIVO. □

The proof follows from the observation that the set $S$ will always be a set that is consistent with the sample in every round of interaction and the fact that a consistent learner cannot produce the same set twice (since the sample given by the oracle for any hypothesized set $R$ is guaranteed to refute $R$), and the fact that there are only finitely many subsets of $\mathcal{P}$.

## 3. Reducing Learning of Sets under Constraints to ICE Learning

In this section, we show how the LSUC problem can be reduced to learning conjunctive formulas in the so-called *ICE learning framework* [34]. In fact, it is enough to reduce the passive LSUC problem to passive ICE-learning as this immediately gives us the reduction of learning sets in the iterative setting to iterative ICE-learning, including the properties of time-complexity and round-complexity (this will be discussed in Section 4).

We first briefly review the ICE learning framework in Section 3.1. In Section 3.2, we describe the reduction of the passive learning of sets under constraints to learning conjunctive formulas in the ICE learning framework.

Before we begin, however, let us fix some notation. First, let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be a finite set of predicates; to ease notation, we assume an ordering on $\mathcal{P}$ and denote the $i$-th element by $p_i$. Next, let $\mathcal{B} = \{b_1, \ldots, b_n\}$ be a set of Boolean variables, again equipped with an ordering such that $b_i$ corresponds to $p_i$, and $\mathcal{BV}_n$ the set of all Boolean vectors of length $n$. Finally, given a conjunctive Boolean formula $\varphi = b_{i_1} \wedge \ldots \wedge b_{i_\ell}$, $i_j \in \{1, \ldots, n\}$ for all $j \in \{1, \ldots, \ell\}$, we say that $\varphi$ *satisfies* a Boolean vector $\vec{v} = (v_1, \ldots, v_n) \in \mathcal{BV}_n$, denoted by $\vec{v} \models \varphi$, if and only if $v_{i_j} = 1$ for all $j \in \{1, \ldots, \ell\}$.

### 3.1 The Passive ICE Learning Framework

The ICE learning framework [34] is a general framework for learning inductive invariants (as in this paper, the ICE learning framework describes a passive and an iterative setting). However, we are here interested in learning conjunctive formulas over Boolean variables and, hence, simplify the description of the framework slightly to better fit our setting.

Roughly speaking, the objective of a learning algorithm in the passive ICE learning framework is to learn a conjunctive formula over $\mathcal{B}$ from positive examples, negative examples, and implications. These examples are given as a so-called ICE sample: an *ICE sample* is a triple $\mathcal{S} = (S_+, S_-, S_\Rightarrow)$ where (a) $S_+ \subseteq \mathcal{BV}_n$ is a set of *positive examples*, (b) $S_- \subseteq \mathcal{BV}_n$ is a set of *negative examples*, and (c) $S_\Rightarrow \subseteq \mathcal{BV}_n \times \mathcal{BV}_n$ is a set of *implications*. We call a conjunctive formula $\varphi$ over $\mathcal{B}$ *consistent with* $\mathcal{S}$ if it satisfies the following three conditions:

1. $\vec{v} \models \varphi$ for all $\vec{v} \in S_+$;
2. $\vec{v} \not\models \varphi$ for all $\vec{v} \in S_-$; and
3. $\vec{v}_1 \models \varphi$ implies $\vec{v}_2 \models \varphi$ for all $(\vec{v}_1, \vec{v}_2) \in S_\Rightarrow$.

Having defined this, we can now define the *passive* ICE learning problem formally:

> *Given an ICE sample $\mathcal{S}$, construct a conjunctive Boolean formula $\varphi$ that is consistent with $\mathcal{S}$.*

The remainder of this section shows how passive ICE learning algorithms can be used to learn sets under constraints, which, in turn, model counterexamples of non-provability.

### 3.2 Reducing passive LSUC to passive ICE Learning of Conjuncts

Recall the passive LSUC problem from Section 2.2, and the notion of when a set $X \subseteq \mathcal{P}$ is consistent with a sample $\mathfrak{S} = (AS, DS, DC)$.

The key idea of our reduction is to translate the sample $\mathfrak{S}$ into an ICE sample $\mathcal{S}$ such that a Boolean formula that is consistent with $\mathcal{S}$ can be translated into a set $X \subseteq \mathcal{P}$ that is consistent with $\mathfrak{S}$. This translation relies on two bijective mappings $\tau$ and $\kappa$. Intuitively, $\tau$ translates between subsets of $\mathcal{P} = \{p_1, \ldots, p_n\}$ and Boolean vectors of length $n$, whereas $\kappa$ translates between subsets of $\mathcal{P}$ and conjunctive formulas over the Boolean variables $\mathcal{B} = \{b_1, \ldots, b_n\}$. Formally, we define $\tau$ and $\kappa$ as follows:

- The mapping $\tau$ maps a set $Y \subseteq \mathcal{P}$ of predicates to the Boolean vector $\vec{v} = (v_1, \ldots, v_n) \in \mathcal{BV}_n$ with

$$v_i = 1 \text{ if and only if } p_i \notin Y.$$

- The mapping $\kappa$ maps a set $Y \subseteq \mathcal{P}$ of predicates to the conjunctive formula $\varphi = \bigwedge_{p_i \in Y} b_i$.

Note that both $\tau$ and $\kappa$ are in fact bijective mappings and, hence, unique inverse mappings $\tau^{-1}$ and $\kappa^{-1}$ exist.

Our reduction now proceeds in three steps:

1. Given the sample $\mathfrak{S} = (AS, DS, DC)$, we translate $\mathfrak{S}$ into the ICE sample $\tau(\mathfrak{S}) = (S_+, S_-, S_\Rightarrow)$ where
   - $S_+ = \{\tau(T_+) \mid T_+ \in AS\}$;
   - $S_- = \{\tau(\mathcal{P} \setminus T_-) \mid T_- \in DS\}$; and
   - $S_\Rightarrow = \{(\tau(\mathcal{P} \setminus U_1), \tau(U_2)) \mid (U_1, U_2) \in DC\}$.
2. We learn a conjunctive Boolean formula $\varphi$ that is consistent with $\tau(\mathfrak{S})$.
3. We return the set $\kappa^{-1}(\varphi)$.

It is left to show that this reduction is indeed correct.

**Theorem 2.** *Let $\mathfrak{S} = (AS, DS, DC)$ be a sample and $X \subseteq \mathcal{P}$ a set of predicates. Then, $X$ is consistent with the sample $\mathfrak{S}$ if and only if $\kappa(X)$ is consistent with the ICE sample $\tau(\mathfrak{S})$.*

*Proof.* See Appendix A. □

In other words, if $\varphi$ is a solution of the ICE learning problem (i.e., $\varphi$ is a conjunctive Boolean formula that is consistent with the ICE sample $\tau(\mathfrak{S})$), then $\kappa^{-1}(\varphi)$ is a set consistent with $\mathfrak{S}$.

## 4. ICE Learners for Conjunctive Boolean Formulas

In this section, we examine existing ICE learners for conjuncts, and build two new ICE learners for conjuncts, and compare and contrast them.

The first learner is an existing learner called HOUDINI [31]. Though HOUDINI is seen as a particular way to synthesize invariants, it is best seen as an ICE learner for conjuncts, as described in previous work by Garg et al [34]. The Houdini algorithm in every round learns the *largest* set of conjuncts that satisfies all positive and implication samples (and hence can never come across a negative sample), and hence learns, in the iterative setting, the invariant that can be expressed using the *largest* number of conjuncts. However, the time it spends in each round is *polynomial* and most importantly, it takes only a linear number of rounds to learn a target concept in an iterative setting.

The property of the Houdini algorithm that makes it converge in at most $n$ rounds (where $n = |\mathcal{P}|$) is very attractive, and as we shall show in experiments later, is very advantageous in practice when learning from a large set of predicates.

However, as we will show in the experiments, Houdini learns in each round the *largest* set of conjuncts, and hence usually ends up with invariants that consist of a large number of conjuncts. In fact, as described below, Houdini is *independent* of negative samples, and hence independent of the assertions in a program— it learns the tightest inductive invariant expressible as a set of conjuncts (which always exists). The downside of Houdini is that the large invariants it learns is often too complex to be human-readable or understandable.

| Learning Algorithm | Property driven | Learning complexity in each round | Maximum # Rounds | Final # of conjuncts |
|---|---|---|---|---|
| Houdini | No | Polynomial | $O(n)$ | Largest set of conjuncts |
| ICE-CS | Yes | NP-complete | $O(2^n)$ | Minimal set of conjuncts |
| ICE-Haussler | Yes | Polynomial | $O(2^n)$ | Bias towards minimal set of conjuncts |

This motivates the two other learners ICE-CS and ICE-Haussler that we develop, both new and property-driven, and contributions of this paper (see table above). The ICE-CS learner learns conjunctions for ICE samples using an SMT solver, and in fact, finds the *smallest* set of conjuncts that satisfies the sample, in sharp contrast to Houdini which finds the largest set. Consequently, in an iterative setting, this learner is guaranteed to find a smallest invariant expressed as a set of conjuncts. However, ICE-CS, by the very fact that it finds smallest formulae, can take an exponential number of rounds.

The ICE-Haussler learning algorithm is a *machine-learning* algorithm inspired by Haussler's algorithm [41] for positive/negative examples, and we provide a natural extension of it to the ICE setting. It is a greedy algorithm that runs in polynomial time, and strives to learn a small set of conjuncts, though not necessarily minimal. It also takes an exponential number of rounds to converge, but has an advantage in that it produces small invariants by examining statistical measures of how often predicates appear true or false in samples.

As we will show in the experimental evaluation, the Houdini algorithm works best in finding an adequate invariant but learns large invariants. However, we will show that by *combining Houdini and the ICE-CS/ICE-Haussler* learners, we can build invariant synthesis engines that both find invariants fast and find small invariants.

We now describe the three ICE learners.

### 4.1 Houdini as an ICE learner

Houdini [31] is a learning algorithm for conjunctive formulas over an a priori given set $\mathcal{B} = \{b_1, \ldots, b_n\}$ of Boolean variables. Given an ICE sample $\mathcal{S} = (S_+, S_-, S_\Rightarrow)$, Houdini computes the largest conjunctive formula $\varphi$ in terms of the number of Boolean variables occurring in $\varphi$ (i.e., the semantically strongest conjunctive formula) that is consistent with $\mathcal{S}$ in the following way. First, it computes the largest conjunction $\varphi$ that is consistent with the positive examples (i.e., $\vec{v} \models \varphi$ for all $\vec{v} \in S_+$); note that this conjunction is unique. Next, Houdini checks whether the implications are satisfied. If this is not the case, then we know for each non-satisfied implication $(\vec{v}_1, \vec{v}_2) \in S_\Rightarrow$ that $\vec{v}_2$ has to be classified positively because $\vec{v}_1$ belongs to every set that includes $S_+$. Hence, Houdini adds all these configurations $\vec{v}_2$ to $S_+$, resulting in a new set $S'_+$. Subsequently, it constructs the largest conjunction $\varphi'$ that is consistent with the positive examples in $S'_+$ (i.e., $\vec{v} \models \varphi'$ for all $\vec{v} \in S'_+$). Houdini repeats this procedure until it arrives at the largest conjunctive formula $\varphi^*$ that is consistent with $S_+$ and $S_\Rightarrow$ (again, note that this set is unique). Finally, Houdini checks whether each negative example violates $\varphi^*$ (i.e., $\vec{v} \not\models \varphi^*$ for all $\vec{v} \in S_-$). If this is the case, $\varphi^*$ is the largest conjunctive formula over $\mathcal{B}$ that is consistent with $\mathcal{S}$; otherwise, clearly no consistent conjunctive formula exists.

### 4.2 A learner based on constraint solving

We now build a learner that learns the syntactically smallest conjunction that satisfies an ICE sample. Since this problem is clearly NP-hard (even learning the smallest conjunction for positive/negative samples is NP-complete [41]), we use a constraint-solver (over Boolean and integer arithmetic) to solve this problem.

We develop a learning algorithm that reduces the problem of finding a consistent conjunctive formula with the minimal number of conjuncts to a series of satisfiability problems. More specifically, given an ICE sample $\mathcal{S}$ and $k \in \{0, \ldots, n\}$, we construct a formula $\psi_{\mathcal{S},k}(\vec{x})$ that is satisfiable if and only if there exists a conjunctive Boolean formula with $k$ conjuncts that is consistent with $\mathcal{S}$; in addition, $\psi_{\mathcal{S},k}$ is designed such that a model $\mathfrak{M} \models \psi_{\mathcal{S},k}$ carries sufficient information to construct a conjunction $\varphi_{\mathfrak{M}}$ that is consistent with $\mathcal{S}$ and comprises exactly $k$ conjuncts. By iterating over successively larger values of $k$ (or by using a binary search), we can easily identify the smallest $k$ for which $\psi_{\mathcal{S},k}$ is satisfiable and use the corresponding model to derive a consistent conjunction with the least number of conjuncts. On the other hand, if even $\psi_{\mathcal{S},n}$ is unsatisfiable, we can report that no consistent conjunction over $\mathcal{B}$ exists that is consistent with the ICE sample.

Let us fix a set of Boolean *variables* $\vec{x}$, where each $x_i$ indicates whether the variable $b_i$ should occur in the final conjunction (i.e., if $x_i$ is set to *true*, then $b_i$ occurs in the conjunction). Moreover, let us fix the following formula macro:

$$t(b_1, \ldots, b_n, x_1, \ldots, x_n) \coloneqq \textstyle\bigwedge_{i=1}^{n} x_i \Rightarrow b_i$$

where $b_i \in \mathcal{B}$ for $i \in \{1, \ldots, n\}$ are Boolean variables. For any valuation $\vec{v}$ of $\vec{b}$, $t[\vec{v}/\vec{b}, \vec{x}]$ demands that the valuation $\vec{v}$ satisfy the conjunct defined by $\vec{x}$.

Given an ICE sample $\mathcal{S} = (S_+, S_-, S_\Rightarrow)$, the learner constructs the following Boolean formula demanding that the conjunct defined by $\vec{x}$ satisfy the ICE-sample:

$$\psi_{\mathcal{S}} \coloneqq \bigwedge_{\vec{v} \in S_+} t[\vec{v}/\vec{b}] \;\wedge\; \bigwedge_{\vec{v} \in S_-} \neg t[\vec{v}/\vec{b}] \;\wedge\; \bigwedge_{(\vec{v}_1, \vec{v}_2) \in S_\Rightarrow} t[\vec{v}_1/\vec{b}] \Rightarrow t[\vec{v}_2/\vec{b}].$$

Note that the free variables of the formula above are $x_1, \ldots, x_n$.

Next, the learner constructs a formula $\psi_k$ that enforces that exactly $k$ of the variables $x_i$ are set to *true*, modeling the restriction that conjunction has exactly $k$ conjuncts. There are several ways to impose this constraint, and we use a straightforward formulation in integer linear arithmetic:

$$\psi_k := \left(\Sigma_{i=1}^{n} ite(x_i, 1, 0)\right) = k$$

where $ite(c,t,f)$ corresponds to the if-then-else term "if $c$ then $t$ else $f$".

The learner checks whether the formula $\psi_{\mathcal{S},k} := \psi_{\mathcal{S}} \wedge \psi_k$ is satisfiable for increasing values of $k$ until the formula becomes satisfiable, say with model $\mathfrak{M}$, or $k = n+1$ is reached. In the first case, the learner returns the conjunction

$$\varphi := \bigwedge_{i \mid \mathfrak{M}(x_i)=true} b_i$$

In the latter case, the learner reports that there exists no conjunct satisfying the ICE sample.

### 4.3 An ICE Learner based on Haussler's algorithm

Haussler's algorithm [41] is a classical learning algorithm that learns small conjunctive concepts from positive and negative examples. In this section, we build a new adaption of Haussler's algorithm to learn small conjunctive Boolean formulas in an ICE setting.

By a reduction to the minimum set cover problem, Haussler shows that learning the *smallest* conjunctive hypothesis consistent with a sample is NP-hard [41]. (This is the reason why the learner in the previous section, which learns the smallest conjunctive hypothesis, uses an SMT solver.) Haussler's algorithm approximates the smallest conjunctive hypothesis in polynomial time using a greedy heuristic. Intuitively, the algorithm starts with the largest (in terms of the number of predicates) conjunctive hypothesis $Q$ which is consistent with all positive examples. It does so by starting with a conjunction of all predicates and iteratively knocking those off that are marked `false` in some positive example in the sample. Once $Q$ is obtained, Haussler's algorithm chooses a small subset of predicates in $Q$ such that their conjunction eliminates all negative examples. The algorithm uses a greedy heuristic for choosing this small subset. It starts with the predicate that eliminates the largest number of negative examples (formally, we say that an example $\vec{u}$ is eliminated by predicate $b_i$ if $\vec{u} \not\models b_i$). The examples eliminated are removed and the process of choosing predicates continues till all negative examples have been eliminated. The algorithm finally returns the conjunction of all chosen predicates as the learned hypothesis $H$.

Obviously, since predicates in $H$ are a subset of predicates in $Q$, $H$ continues to be consistent with all positive examples. In addition, $H$ eliminates all negative examples in the sample (from the choice of predicates that constitute $H$). Hence, the learned hypothesis $H$ will be consistent with the sample.

The adaptation of Haussler's algorithm to ICE samples is presented as Algorithm 1. We first construct the largest conjunctive formula $Q$ that is consistent with all positive *and implication examples* in the sample. This construction follows the algorithm detailed in Section 4.1. Subsequently, as in the original Haussler's algorithm, we greedily choose a subset of predicates in $Q$ which eliminate all negative examples. For every predicate $b_i$, we compute *neg-eliminated*$(b_i)$, which is the set of negative examples eliminated by $b_i$. Also, for every implication example $(\vec{u},\vec{v})$ where $\vec{v}$ is eliminated by $b_i$, $\vec{u}$ effectively becomes a negative example and needs to be eliminated if $b_i$ is chosen in the final set of predicates and if it does not eliminate $\vec{u}$ already. All such implication points constitute *neg-added*$(b_i)$ (we assume in the presentation of Algorithm 1 that $S_{\Rightarrow}$ is transitively closed with respect to implications, i.e., if $(\vec{u},\vec{v}) \in S_{\Rightarrow}$ and $(\vec{v},\vec{r}) \in S_{\Rightarrow}$ then $(\vec{u},\vec{r}) \in S_{\Rightarrow}$; otherwise, the set *neg-added*$(b_i)$ would need to be computed recursively). Finally, in every round we choose $b_i$ that minimizes the number of negative examples remaining that need to be eliminated.

**Input**: $\mathcal{B} = \{b_1, \ldots, b_n\}$, $\mathcal{S} = (S_+, S_-, S_{\Rightarrow})$
$Q \leftarrow \{b_{i_1}, \ldots, b_{i_q}\}$ s.t. $\bigwedge_{i=1}^{q} b_{i_j}$ is the largest conjunctive formula consistent with $S_+$ and $S_{\Rightarrow}$;
$H \leftarrow$ *True* ;
**while** $S_-$ *is non empty* **do**
  **foreach** $b_i \in Q$ **do**
    *neg-eliminated*$(b_i) \leftarrow$ (examples in $S_-$ eliminated by $b_i$) ;
    *neg-added*$(b_i) \leftarrow \{\vec{u} \mid (\vec{u},\vec{v}) \in S_{\Rightarrow}$
      *and* $\vec{u} \models H \wedge b_i$ *and* $\vec{v} \not\models H \wedge b_i\}$ ;
    $S_-(b_i) \leftarrow S_- \setminus$ *neg-eliminated*$(b_i) \cup$ *neg-added*$(b_i)$ ;
  **end**
  Select $i$ with smallest set cardinality $|\, S_-(b_i) \,|$ ;
  $H \leftarrow H \wedge b_i$ ;
  $Q \leftarrow Q \setminus \{b_i\}$ ;
  $S_- \leftarrow S_-(b_i)$ ;
**end**
**return** $H$ ;

**Algorithm 1:** ICE adaptation of Haussler's algorithm [41]

From the stopping criterion of the algorithm we know that in the last round no new implication point is added to the negative sample. This implies that if $H$ is the final hypothesis then for all implication examples $(\vec{u},\vec{v})$ if $\vec{u} \models H$ then $\vec{v} \models H$, i.e., the final hypothesis is consistent with all implications. Also, as we argued in the original Haussler's algorithm, the final $H$ is also consistent with all negative and positive examples. This proves the consistency of the learned hypothesis with respect to the sample. As for termination, notice that the cardinality of $Q$ decreases in each iteration of the while-loop. Further, if $Q$ gets empty and the loop terminates and $S_-$ is still nonempty, then it is easy to see that there is no conjunctive concept that satisfies the sample, and we can terminate and report that no formula exists.

## 5. Learning Invariants that Aid Natural Proofs for Heap Reasoning

We develop, in this section, an instantiation of our learning framework for synthesizing invariants that aid verification based on natural proofs for heap reasoning, as developed in the work [61, 65].

We first give some background on the separation logic DRYAD and natural proofs for DRYAD, which is a sound but incomplete verification procedure. Then we give the verification framework we build that pairs a verification oracle based on natural proofs with black-box learners that learn from non-provability information encoded using constraints on sets.

### 5.1 Background: Natural proofs and DRYAD

DRYAD [61, 65] is a dialect of separation logic that comes with a heaplet semantics and allows expressing second order properties such as a list segment or a list using recursive functions and predicates. The syntax of DRYAD is a standard separation logic syntax with a few restrictions, such as disallowing negations inside recursive definitions and in sub-formulas connected by spatial conjunction (please refer to [61] for more details about the DRYAD syntax). DRYAD is expressive enough to state a variety of data-structures (singly and doubly linked lists, sorted lists, binary search trees, AVL trees, maxheaps, treaps, etc.) and recursive definitions over them that map to numbers (length, height, etc.) as well as data stored within the heaps (the multiset of keys stored in lists, trees, and so on).

The main difference between DRYAD and classical separation logic lies in the semantics. Unlike classical separation logic, the heap domain on which any DRYAD formula holds true is syntactically determined. With this semantics, one can construct the heap domain for any DRYAD formula through one bottom-up traversal of the syntactic parse tree of the formula (see [65] for details).

Natural proofs [61, 65] are a sound but incomplete strategy for verifying such programs. The first step in this converts all predicates and functions in DRYAD to *classical logic*. This translation introduces *heaplets* (modeled as sets of locations) explicitly in the logic, and furthermore, introduces the assertions that demand that the accesses of each method are contained in the heaplet implicitly defined by its precondition (taking into account newly allocated or freed nodes), and that at the end of the program, the modified heaplet precisely matches the implicit heaplet defined by the post-condition.

The second step the natural proof verifier does is to perform *transformations* on the program and translate it to BOOGIE [31], an intermediate verification language that handles proof obligations using automatic theorem provers (typically SMT solvers). VCDRYAD extends VCC [19] to perform several natural proof transformations that essentially perform three tasks: (a) abstract all recursive definitions on the heap using uninterpreted functions but introduce unfolding the recursive definitions at every place in the code where locations are dereferenced, (b) model heaplets and other sets using a decidable theory of maps, (c) insert *frame reasoning* explicitly in the code that allows the verifier to know certain properties continue to hold across a heap update (or function call) using the heaplet that is modified.

The resulting program is a BOOGIE program with no recursive definitions and where all verification conditions would be in decidable logics (caveat: there is some quantification introduced byVCC itself to define the memory model and semantics of C, but this does not typically derail decidable reasoning). Consequently, the program can be verified if supplied with correct inductive loop-invariants and adequate pre/post conditions.

### 5.2 Extracting Non-provability Information from Natural Proof Verifiers

While natural proofs are a sound (but incomplete) approach to verify programs against heap properties, they can also provide non-provability counterexamples. They transform verification conditions involving undecidable logics that include recursion (amongst others) to decidable logics such that the latter's validity implies the validity of the former. This aspect of natural proofs is the crucial property that we use to generate counterexamples. While failure to find a natural proof cannot be used to produce true counterexamples, we will show how a model showing the failure to find a natural proof can be used to provide *constraints on the set of conjuncts in the invariant* as described in Section 2.

First, let us fix a set of predicates $\mathcal{P}$ in classical logic such that the space of invariants we want to learn from are conjuncts of formulas from $\mathcal{P}$. The set of predicates typically depends on the data-structures and recursive definitions used in the program and its annotations, and we will describe how we choose this later.

Given a Hoare triple $\{\alpha\}s\{\beta\}$, the natural proof technique fails to prove the validity of the associated verification condition when the associated *abstraction* of the verification condition expressed in a decidable logic fails to be proved valid. In this case, the SMT solver would find that the negation of the verification condition is satisfiable.

To extract non-provability information, we first augment the program snippets $s$ so that we record, using a set of ghost variables $b_1 \ldots b_n$, whether the predicates $p_1, \ldots, p_n$ are true, before and after the execution of $s$. Consequently, when the SMT solver finds the negation of the verification condition to be satisfiable, it would also find a valuation of the variables $b_1 \ldots b_n$ before and after the snippet $s$. We can now use this information to find the constraints on sets.

More precisely, we find the LSUC sample from such a counterexample model as follows. Let $v_1, \ldots, v_n$ denote the valuation of $b_1, \ldots, b_n$ before executing $s$, and let $v'_1, \ldots, v'_n$ denote the valuation of $b_1, \ldots, b_n$ after executing $s$.

- If the Hoare triple is $\{\alpha\}s\{\gamma\}$, where $\gamma$ is the synthesized invariant, then we take the avoid set $T_+$ to be the set of predicates $p_i$ such that $v'_i = \mathit{false}$.
- If the Hoare triple is $\{\gamma\}s\{\alpha\}$, where $\gamma$ is the synthesized invariant, then we take differ set $T_-$ to be the set of predicates $p_i$ such that $v_i = \mathit{true}$.

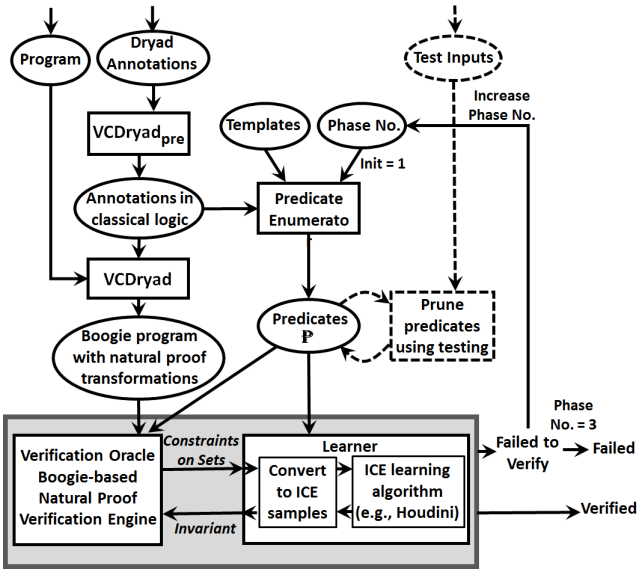


**Figure 1.** Framework for proving programs using natural proofs and invariant synthesis using learning; (components marked with broken lines are optional)

- If the Hoare triple is $\{\gamma\}s\{\gamma\}$, where $\gamma$ is the synthesized invariant, then we form the double constraint $(U_1, U_2)$, where $U_1$ is the set of predicates $p_i$ such that $v_i = true$ and $U_2$ is the set of predicates $p_i$ such that $v_i' = false$.

It is straightforward to see that this accurately gives the non-provability entailed by the counterexample model found by the SMT engine, and the sample meets the criteria of avoid sets, differ sets, and double constraints described in Section 2.1.

The above gives an adequate way to report non-provability information, and hence allows us to pair any learner of sets under constraints to obtain an invariant synthesis tool for proving using natural proofs.

### 5.3 A Framework Combining Natural Proofs and Invariant Learning

We now describe our proposed framework that combines natural proofs and black-box invariant learning of conjunctions, as depicted in Figure 1. As an input we take a program $P$ with DRYAD definitions and annotations. We then apply the translation of separation logic constraints to classical logic, from [61, 65], to obtain the program $P$ with purely classical logic annotations, which involve recursive definitions over sets of locations and integers. This program can then be transformed using natural proof transformations that unfold definitions depending on the dereferences the program makes, and abstract recursive definitions to uninterpreted functions, again from [61, 65].

**Choosing predicates that form invariants:**

Second, given the DRYAD definitions of data structures, we generate a set of *predicates*, the conjuncts of subsets of which will form the space of invariants.

Figure 2 shows the class of templates that we consider; it forms a fairly exhaustive combination of various formulas and predicates. Predicates include properties of the store (equality of pointer variables, equality and inequalities between integer variables, etc.), shape properties (singly and doubly linked lists and list segments, sorted lists, trees, BST, AVL, treaps, etc.), predicates involving recursive definitions from data structures to numbers (keys/data stored in a structure, lengths of lists and list segments, height of trees) involving arithmetic relationships and set relationships. Furthermore, there are also predicates involving the heaplets of various structures (with suffix *_heaplet*) that involve set operations, disjointness, and equality. The structures and predicates are extensible, of course, to any recursive definition expressed in DRYAD .

Besides these general purpose predicates, we also consider program-specific predicates that occur in the pre- and post-conditions of programs.

Furthermore, as an optimization, we divide the predicates into three phases, roughly ordered by their complexity (see Figure 2). When verifying a program, we first try the Phase 1 predicates, and if there is no invariant that proves the program, we add the Phase 2 predicates, and finally add the Phase 3 predicates. This allows us to prove programs with simpler invariants faster.

**Learning Invariants:**

The program verifier and the learner that learns sets of predicates under constraints are then paired in an iterative learning loop to find the invariant (see Figure 1). Using the reduction of learning sets under constraints to ICE-learning of conjuncts (Section 3), we can use any of the ICE-learners for conjuncts we developed in Section 4.

**Initial Pruning of Predicates with Testing**

We also incorporate in our framework an *optional* component that allows the user to create certain test inputs, which we can use to prune predicates and hence prove the program faster.

As we will show in the evaluation section, it is not uncommon to have hundreds or even thousands of candidate predicates, most of which are not invariants in the program, and a learning algorithm that uses only a verification oracle will have to use constraint solvers to refute them. However, if we have a few test inputs to the program that satisfy the program's preconditions, then we can execute the program on these inputs, and eliminate all predicates that are falsified at least once when the program reaches the annotation we are trying to synthesize. This is sound because the predicates in a conjunctive invariant obviously cannot contain a predicate that is violated in a test. This testing-based pruning technique is similar to Daikon [30] and can be seen as a Daikon-style

$x, y, x_1, x_2 \in PointerVars \quad \vec{x}, \vec{y}, \vec{z} \in PointerVars^* \quad i, j \in IntegerVars \cup \{0, \mathsf{IntMax}, \mathsf{IntMin}\} \quad pf \in PointerFields \quad key,\ df \in DataFields$

$$
\begin{array}{rcl}
listshape(\vec{x}) & := & \mathsf{LinkedList}(x_1) \mid \mathsf{DoublyLinkedList}(x_1) \mid \mathsf{SortedLinkedList}(x_1) \\
 & \mid & \mathsf{LinkedListSeg}(x_1,x_2) \mid \mathsf{DoublyLinkedListSeg}(x_1,x_2) \mid \mathsf{SortedLinkedListSeg}(x_1,x_2) \\
treeshape(x) & := & \mathsf{BST}(x) \mid \mathsf{AVLtree}(x) \mid \mathsf{Treap}(x) \\
shape(\vec{x}) & := & listshape(\vec{x}) \mid treeshape(\vec{x}) \\
size(\vec{x}) & := & listshape\_\mathsf{length}(\vec{x}) \mid treeshape\_\mathsf{height}(\vec{x})
\end{array}
$$

| | | | |
|---|---|---|---|
| *Phase* 1 | $shape(\vec{x})$ | $x = \mathsf{nil}$ | $x = y$ |
| | $x \in shape\_\mathsf{heaplet}(\vec{y})$ | $x \neq \mathsf{nil}$ | $x \neq y$ |
| | $x \notin shape\_\mathsf{heaplet}(\vec{y})$ | $x.pf = \mathsf{nil}$ | $x.pf = y$ |
| | $shape\_\mathsf{heaplet}(\vec{x}) \cap shape\_\mathsf{heaplet}(\vec{y}) = \emptyset$ | $x.pf \neq \mathsf{nil}$ | $x.pf \neq y$ |
| *Phase* 2 | $i \in shape\_key\_\mathsf{set}(\vec{x})$ | $x.df \leq y.df$ | $x.df \leq i$ |
| | $i \notin shape\_key\_\mathsf{set}(\vec{x})$ | $x.df \geq y.df$ | $x.df \geq i$ |
| | $shape\_key\_\mathsf{set}(\vec{x}) = shape\_key\_\mathsf{set}(\vec{y})$ | $x.df = y.df$ | $x.df = i$ |
| | $shape\_key\_\mathsf{set}(\vec{x}) = shape\_key\_\mathsf{set}(\vec{y}) \cup shape\_key\_\mathsf{set}(\vec{z})$ | $x.df \neq y.df$ | $x.df \neq i$ |

| | | | |
|---|---|---|---|
| *Phase* 3 | $size(\vec{x}) = i$ | $size(\vec{x}) \geq i$ | $shape\_key\_\mathsf{set}(\vec{x}) \leq_{\mathsf{set}} \{i\}$ |
| | $size(\vec{x}) = i - j$ | $size(\vec{x}) \leq i$ | $shape\_key\_\mathsf{set}(\vec{x}) \leq_{\mathsf{set}} \{y.df\}$ |
| | $size(\vec{x}) - size(\vec{y}) = i$ | $shape\_key\_\mathsf{set}(\vec{x}) \geq_{\mathsf{set}} shape\_key\_\mathsf{set}(\vec{y})$ | $shape\_key\_\mathsf{set}(\vec{x}) \geq_{\mathsf{set}} \{i\}$ |
| | $size(\vec{x}) - size(\vec{y}) = i - j$ | $shape\_key\_\mathsf{set}(\vec{x}) \leq_{\mathsf{set}} shape\_key\_\mathsf{set}(\vec{y})$ | $shape\_key\_\mathsf{set}(\vec{x}) \geq_{\mathsf{set}} \{y.df\}$ |

**Figure 2.** Templates for predicates. The operator $\leq_{\mathsf{set}}$ denotes comparison between integer sets, where $A \leq_{\mathsf{set}} B$ if and only if $\forall x \in A. \forall y \in B.\ x \leq y$. The operator $\geq_{\mathsf{set}}$ is also similarly defined.

synthesis using tests to find initial *likely invariants* before proceeding with verification.

## 6. Evaluation

**Implementation:** We implemented the verification framework (described in Section 5). We use VCDryad [61], a Boogie-based natural proofs engine that builds on VCC for verifying C programs, as the verification oracle. By instrumenting the programs in Boogie, we implemented the reduction from learning sets under constraints to ICE-learning. We could then use the Houdini-based ICE learner already available in Boogie [4, 31]. We implemented the ICE-CS learner (using Z3 as constraint solver) and ICE-Haussler learning algorithms, and interfaced them with Boogie as well.

We also implemented the testing-based-pruning of predicates, which take in test inputs to programs and a class of predicates, and automatically instruments them to prune the predicates that do not hold in some execution of a test.

**Benchmarks:** Our benchmarks are primarily drawn from the benchmarks used in evaluating VCDRYAD [61], since this was our verification oracle. This benchmark suite consists of various list and tree manipulation programs, including singly-linked lists, doubly-linked lists, sorted lists, binary search trees, AVL trees, and treaps. The programs were taken from GNU C library, OpenBSD operating system, ExpressOS [56], a secure operating system for mobile devices, and another two benchmarks of other verification engines, namely GRASShopper [62] and AFWP [46]. The specifications for these programs are generally for full functional correctness properties, such as preserving or altering shapes of data structures, inserting or deleting keys, filtering or finding elements, and sorted-ness of elements in structures. The specifications hence involve separation logic with arithmetic as well as recursive definitions that compute numbers (like lengths and heights) and data-aggregating recursive functions (like multisets of keys stored in data-structures), and complex data-structures that combine these properties (like binary search trees and treaps).

From this benchmark suite, we first picked all programs that contained iterative loops, and *erased* the user-provided loop invariant, and used our framework to find an adequate invariant. We also selected some of the programs that were purely recursive, where the contract for the function had been strengthened to make the verification succeed. We *weakened* these contracts to only state the specification (typically by removing formulas in the post-conditions of recursively-called functions), and introduced annotation holes instead, in order to use our framework. Since our goal is to evaluate our learning technique which calls the verification oracle many times (typically 20 to 30 times), we did not consider programs that take more than 100 seconds to just verify it with a user-specified inductive invariant. Our tool hence was used in both learning invariants for iterative loops as well as strengthening post-conditions for recursive programs.

**Experimental Results** There are a large number of combinations of using the learning algorithms we develop in this paper:

1. We can use an initial testing-based pruning or not.
2. We can use any of the three learners— Houdini, ICE-CS, or ICE-Haussler
3. We can choose to use ICE-CS or ICE-Haussler *after* Houdini, in order to synthesize a smaller invariant once a larger invariant has been found.

| Program | #Candidate Pred | #Pred after TBP | Houdini | | | | ICE-Haussler post Houdini | | | Total Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Max Phase | #Rounds | \|Inv\| | Time (s) | #Rounds | \|Inv\| | Time (s) | |
| **glib/gslist.c Singly Linked List** | | | | | | | | | | |
| g-slist-copy | 428 | | 2 | 105 | 130 | 33.0 | 64 | **3** | 24.6 | 57.6 |
| g-slist-find | 22 | | 1 | 14 | 4 | 1.2 | 9 | **3** | 2.9 | 4.1 |
| g-slist-free | 22 | | 1 | 15 | 1 | 1.6 | 6 | **1** | 2.1 | 3.7 |
| g-slist-index | 261 | 74 | 3 | 6 | 60 | 4.5 | 77 | **6** | 29.2 | 33.7 |
| g-slist-insert | 188 | 76 | 1 | 9 | 42 | 11.6 | 47 | **7** | 46.3 | 57.9 |
| g-slist-insert-before | 385 | 248 | 1 | 83 | 90 | 52.8 | 24 | **6** | 51.0 | 103.8 |
| g-slist-insert-sorted | 556 | | 2 | 209 | **138** | 353.0 | – | – | 600.0 | 953.0 |
| g-slist-last | 22 | | 1 | 15 | 6 | 1.6 | 9 | **4** | 2.9 | 4.5 |
| g-slist-length | 56 | 18 | 3 | 4 | 10 | 3.8 | 8 | **1** | 2.2 | 6.0 |
| g-slist-nth | 92 | 31 | 3 | 5 | 15 | 3.8 | 11 | **5** | 3.5 | 7.3 |
| g-slist-nth-data | 372 | 99 | 3 | 9 | 67 | 4.5 | 25 | **6** | 8.8 | 13.3 |
| g-slist-position | 180 | | 3 | 38 | 17 | 10.8 | 24 | **5** | 6.1 | 16.9 |
| g-slist-remove | 188 | 56 | 1 | 3 | 52 | 1.4 | 85 | **6** | 41.2 | 42.6 |
| g-slist-remove-all | 428 | 122 | 2 | 23 | 25 | 13.2 | 39 | **7** | 21.3 | 34.5 |
| g-slist-remove-link | 385 | 122 | 1 | 19 | **65** | 4.8 | – | – | 600.0 | 604.8 |
| g-slist-reverse | 129 | | 2 | 65 | 10 | 14.5 | 10 | **5** | 3.3 | 17.8 |
| g-slist-sort-merge | 1127 | 299 | 1 | 36 | 162 | 123.8 | 4 | **0** | 14.1 | 137.9 |
| **GRASS-hopper Singly Linked List** | | | | | | | | | | |
| sl-concat | 75 | 35 | 1 | 9 | 20 | 1.6 | 26 | **5** | 6.9 | 8.5 |
| sl-copy | 75 | 22 | 1 | 3 | 17 | 1.9 | 6 | **2** | 2.1 | 4.0 |
| sl-dispose | 22 | 6 | 1 | 2 | 1 | 1.6 | 6 | **1** | 2.0 | 3.6 |
| sl-filter | 188 | 56 | 1 | 13 | 19 | 4.3 | 28 | **6** | 14.5 | 18.8 |
| sl-insert | 75 | 19 | 1 | 3 | 15 | 1.2 | 11 | **4** | 3.3 | 4.5 |
| sl-remove | 22 | | 1 | 15 | 6 | 1.4 | 13 | **4** | 3.3 | 4.7 |
| sl-reverse | 75 | 12 | 1 | 2 | 8 | 1.3 | 10 | **3** | 2.9 | 4.2 |
| sl-traverse | 22 | 7 | 1 | 4 | 4 | 1.2 | 8 | **0** | 1.3 | 2.5 |
| **glib/glist.c Doubly Linked List** | | | | | | | | | | |
| g-list-find | 50 | | 2 | 18 | 7 | 2.8 | 9 | **4** | 2.4 | 5.2 |
| g-list-free | 22 | | 1 | 15 | 1 | 1.9 | 6 | **1** | 2.1 | 4.0 |
| g-list-index | 261 | | 3 | 64 | 63 | 13.6 | 70 | **6** | 36.1 | 49.7 |
| g-list-last | 22 | | 1 | 15 | 6 | 1.5 | 9 | **4** | 2.8 | 4.3 |
| g-list-length | 92 | | 3 | 27 | 14 | 5.4 | 8 | **1** | 2.5 | 7.9 |
| g-list-nth | 92 | | 3 | 28 | 15 | 5.4 | 14 | **5** | 4.8 | 10.2 |
| g-list-nth-data | 372 | | 3 | 102 | 70 | 20.9 | 25 | **6** | 11.4 | 32.3 |
| g-list-position | 180 | | 3 | 37 | 17 | 9.2 | 20 | **5** | 6.4 | 15.6 |
| g-list-reverse | 188 | | 1 | 58 | 4 | 9.3 | 4 | **0** | 1.5 | 10.8 |
| **GRASS-hopper Doubly Linked List** | | | | | | | | | | |
| dl-concat | 75 | | 1 | 29 | 20 | 2.3 | 23 | **5** | 8.4 | 10.7 |
| dl-copy | 75 | | 1 | 34 | 17 | 3.7 | 18 | **4** | 12.8 | 16.5 |
| dl-dispose | 188 | | 1 | 56 | 12 | 11.4 | 6 | **1** | 2.4 | 13.8 |
| dl-filter | 188 | | 1 | 75 | 19 | 9.8 | 43 | **7** | 29.9 | 39.7 |
| dl-insert | 75 | | 1 | 34 | 26 | 2.4 | 14 | **4** | 6.8 | 9.2 |
| dl-remove | 22 | | 1 | 15 | 6 | 1.5 | 13 | **4** | 4.5 | 6.0 |
| dl-reverse | 75 | | 1 | 40 | 8 | 3.8 | 10 | **3** | 3.9 | 7.7 |
| dl-traverse | 22 | | 1 | 14 | 4 | 1.2 | 4 | **0** | 1.4 | 2.6 |
| **Sorted Linked List** | | | | | | | | | | |
| sorted-insert-iter | 219 | | 2 | 66 | **53** | 202.5 | – | – | 600.0 | 802.5 |
| find-last-sorted | 40 | | 2 | 19 | 9 | 3.6 | 18 | **5** | 5.9 | 9.5 |
| reverse-sorted | 102 | | 2 | 35 | 13 | 14.8 | 17 | **5** | 9.6 | 24.4 |

| Program | #Candidate Pred | #Pred after TBP | Houdini | | | | ICE-Haussler post Houdini | | | Total Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Max Phase | #Rounds | \|Inv\| | Time (s) | #Rounds | \|Inv\| | Time (s) | |
| **AFWP Singly Linked and Doubly Linked List** | | | | | | | | | | |
| SLL-create | 5 | 3 | 1 | 3 | 1 | 1.1 | 6 | **1** | 1.8 | 2.9 |
| SLL-delete | 385 | 122 | 1 | 20 | **90** | 4.3 | – | – | 600.1 | 604.4 |
| SLL-delete-all | 22 | | 1 | 14 | 1 | 3.3 | 6 | **1** | 2.4 | 5.7 |
| SLL-filter | 75 | 15 | 1 | 2 | 13 | 1.3 | 30 | **6** | 8.9 | 10.2 |
| SLL-find | 188 | 92 | 1 | 25 | 54 | 3.1 | 11 | **2** | 2.7 | 5.8 |
| SLL-insert | 219 | 141 | 2 | 38 | 25 | 26.9 | 101 | **9** | 167.6 | 194.5 |
| SLL-last | 75 | 15 | 1 | 2 | 13 | 1.2 | 24 | **5** | 5.7 | 6.9 |
| SLL-merge | 2040 | 432 | 2 | 55 | 200 | 124.9 | 258 | **16** | 502.1 | 627.0 |
| SLL-reverse | 75 | 12 | 1 | 2 | 8 | 1.2 | 10 | **3** | 2.7 | 3.9 |
| DLL-fix | 12 | | 1 | 3 | 6 | 1.1 | 6 | **1** | 2.1 | 3.2 |
| **GRASS-hopper Sorted List** | | | | | | | | | | |
| sls-concat | 165 | 59 | 2 | 9 | 23 | 5.2 | 17 | **5** | 8.9 | 14.1 |
| sls-copy | 556 | 219 | 2 | 27 | 114 | 308.9 | 62 | **0** | 246.4 | 555.3 |
| sls-dispose | 22 | | 1 | 14 | 1 | 1.6 | 6 | **1** | 1.6 | 3.2 |
| sls-double-all | 188 | 60 | 1 | 2 | 59 | 1.9 | 9 | **2** | 5.7 | 7.6 |
| sls-filter | 556 | | 2 | 179 | 40 | 134.4 | 119 | **13** | 137.2 | 271.6 |
| sls-insert | 165 | 58 | 2 | 6 | 34 | 5.0 | 65 | **7** | 37.3 | 42.3 |
| sls-merge | 2040 | 434 | 2 | 76 | 119 | 171.4 | 197 | **19** | 422.4 | 593.8 |
| sls-pairwise-sum | 690 | 226 | 1 | 3 | **223** | 3.4 | – | – | 600.0 | 603.4 |
| sls-remove | 556 | | 2 | 181 | 113 | 83.4 | 236 | **13** | 175.5 | 258.9 |
| sls-reverse | 102 | | 2 | 35 | 11 | 17.5 | 11 | **4** | 4.8 | 22.3 |
| sls-split | 165 | 51 | 2 | 7 | 27 | 18.1 | 129 | **13** | 135.9 | 154.0 |
| sls-traverse | 22 | | 1 | 14 | 2 | 1.3 | 4 | **0** | 1.1 | 2.4 |
| **Trees** | | | | | | | | | | |
| bst-find | 23 | | 3 | 6 | 9 | 1.4 | 16 | **4** | 9.7 | 11.1 |
| bst-insert | 68 | | 3 | 29 | **16** | 79.0 | – | – | 600.0 | 679.0 |
| bst-delete | 68 | | 3 | 17 | **11** | 269.3 | – | – | 600.0 | 869.3 |
| avl-find | 19 | | 3 | 5 | 11 | 1.1 | 29 | **4** | 15.2 | 16.3 |
| avl-insert | 72 | | 3 | 26 | **14** | 125.6 | – | – | 600.0 | 725.6 |
| avl-delete | 72 | | 3 | 19 | **5** | 358.1 | – | – | 600.0 | 958.1 |
| treap-find | 25 | | 3 | 6 | 11 | 1.7 | 64 | **6** | 110.7 | 112.4 |
| treap-delete | 80 | | 3 | 20 | **13** | 958.9 | – | – | 600.0 | 1558.9 |
| traverse-tree-inorder | 9 | | 3 | 6 | 3 | 1.1 | 2 | **1** | 1.6 | 2.7 |
| traverse-tree-postorder | 9 | | 3 | 6 | 3 | 1.1 | 2 | **1** | 2.3 | 3.4 |
| traverse-tree-preorder | 9 | | 3 | 6 | 3 | 1.1 | 2 | **1** | 5.1 | 6.2 |
| **ExpressOS Examples** | | | | | | | | | | |
| bsd-squeue-insert-head | 5 | | 1 | 3 | 2 | 10.6 | 1 | **1** | 3.2 | 13.8 |
| bsd-squeue-insert-tail | 5 | | 1 | 3 | 3 | 21.8 | 1 | **1** | 1.2 | 23.0 |
| bsd-squeue-remove-head | 12 | | 1 | 5 | 4 | 5.3 | 1 | **1** | 1.1 | 6.4 |
| memory-region-init | 7 | | 1 | 5 | 4 | 1.0 | 1 | **1** | 0.9 | 1.9 |
| memory-region-split | 24 | | 1 | 9 | 6 | 4.5 | 1 | **1** | 1.3 | 5.8 |
| memory-region-find | 24 | | 1 | 16 | 2 | 1.0 | 2 | **1** | 1.1 | 2.1 |
| memory-region-insert | 51 | | 1 | 16 | 3 | 1.2 | 2 | **1** | 1.2 | 2.4 |

**Table 1.** Experimental results for the technique combining testing-based pruning(TBP) followed by Houdini followed by ICE-Haussler. Blank cells under "#Pred after TBP" indicate that testing-based pruning was not performed.

The goal of the experimental evaluation was to evaluate (a) whether the techniques developed in this paper are effective in synthesizing invariants that prove these programs correct, and (b) to compare the efficacy of the various learners.

We evaluate our framework against around 85 routines on all the 12 combinations above, but we present our readings of our experiments by comparing only the pertinent ones. All experiments were performed on a system with an Intel Core i7 2.2 GHz processor and 8GB RAM running 64-bit Windows 10, with a timeout of 1800s (600s per phase).

***Observation#1: Testing based pruning is effective:*** Our first observation is that testing-based pruning is *always* helpful. Writing a few tests (even a couple of arbitrary ones) knocks off a large number of predicates initially, before the learning phase starts. Figure 3(a) compares a subset of benchmarks that take significant time for verification, and compares the time taken by Houdini and Houdini after testing-based pruning. Without testing-based pruning, Houdini was unable to finish 3 of these programs (it crashed), but was able to prove them after testing-based pruning. See also Table 1 which shows the number of predicates pruned using testing. It turns out that testing-based pruning is fast and effective, and useful no matter which learner we use. In the sequel, we will assume this subset of the benchmarks have been subject to testing-based pruning before learning begins.

***Observation#2: Houdini is fastest in finding an invariant but finds large invariants:*** Houdini, due to its guarantee of taking at most $n$ rounds (where $n$ is the number of predicates), proved generally more effective than ICE-CS and ICE-Haussler in finding invariants. Figure 3(b) shows the comparison of Houdini with the *better of ICE-CS and ICE-Hassler* on the benchmarks. In 9 of these, Houdini was able to find an invariant while the other two solvers timed out. Furthermore, as can be seen from the figure, Houdini is in general much faster than the other learners. We refer the reader to Table 1 for the performance of Houdini. Notice that Houdini, though effective, finds extremely *large* conjuncts as invariants, some with hundreds of predicates, and on average 36 predicates! These invariants are clearly not human understandable.

***Observation#3: Using ICE-CS or ICE-Haussler after Houdini significantly reduces the size of invariants*** Though ICE-CS and ICE-Haussler are inferior to Houdini, they are effective post-Houdini to reduce the size of invariants. Table 1 shows the experimental results of applying ICE-Haussler after Houdini (the results for applying ICE-CS after Houdini are similar, and recall that ICE-CS always produces the smallest invariants). As is evident from the table, ICE-Haussler reduces the size of the invariant drastically and is usually less than 5 conjuncts (and on average 4.4 conjuncts). By comparing these to results of using ICE-CS, we found that the invariants learned by ICE-Haussler are in fact close to optimal.

***Observation#4: ICE-Haussler is marginally better than ICE-CS*** Despite ICE-CS producing the smallest invariants, we find, surprisingly, that ICE-Haussler is slightly better than ICE-CS, when used post-Houdini to reduce the size of invariants. Figure 3(c) shows a comparison of times between ICE-CS and ICE-Haussler, both post-Houdini. As the figure shows, ICE-Haussler performs in general a bit better. In fact, it performs better even when we compared the number of rounds— ICE-CS takes more rounds (30% more) than ICE-Haussler, showing that the fact that ICE-CS uses a more expensive passive learning algorithm is not the reason for the decreased efficiency. We do not understand the reason for this entirely but we think that ICE-Haussler has a *statistical* bias in picking predicates that are most often true (while ICE-CS picks any set of minimal predicates) and this may be the reason why learns the most relevant predicates in the invariant faster. Similar observations have been made before by Garg et al [36], where they found machine-learning based learners took less rounds and was more efficient than constraint-solving based learners.

***Conclusion of observations:*** In conclusion, we found that the best algorithm, by far, was to (a) use testing-based reduction, (b) followed by Houdini to learn a large invariant, and (c) to use ICE-Haussler to reduce the invariant to a small human-understandable invariant. Table 1 summarizes the experiments for this setup. The table shows the initial set of predicates generated (for the final phase needed to prove the program), the reduction provided by testing-based pruning, the effectiveness of Houdini in finding an invariant using the pruned predicate set, and the effectiveness of ICE-Haussler in reducing the size of the invariant. Note that ICE-Haussler times-out sometimes and cannot reduce the invariant. The invariant sizes in bold depict the size of the final invariant learned.

The experiments show that learning-based invariant generation against natural proof verification for heap properties is quite effective in proving programs automatically correct.

## References


[1] P. A. Abdulla, L. Holík, B. Jonsson, O. Lengál, C. Q. Trinh, and T. Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. In *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, pages 224–239, 2013.

[2] A. Albarghouthi, J. Berdine, B. Cook, and Z. Kincaid. Spatial interpolants. In J. Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 634–660. Springer, 2015. ISBN 978-3-662-46668-1.

[3] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: smt-based abstraction for arrays with inter-

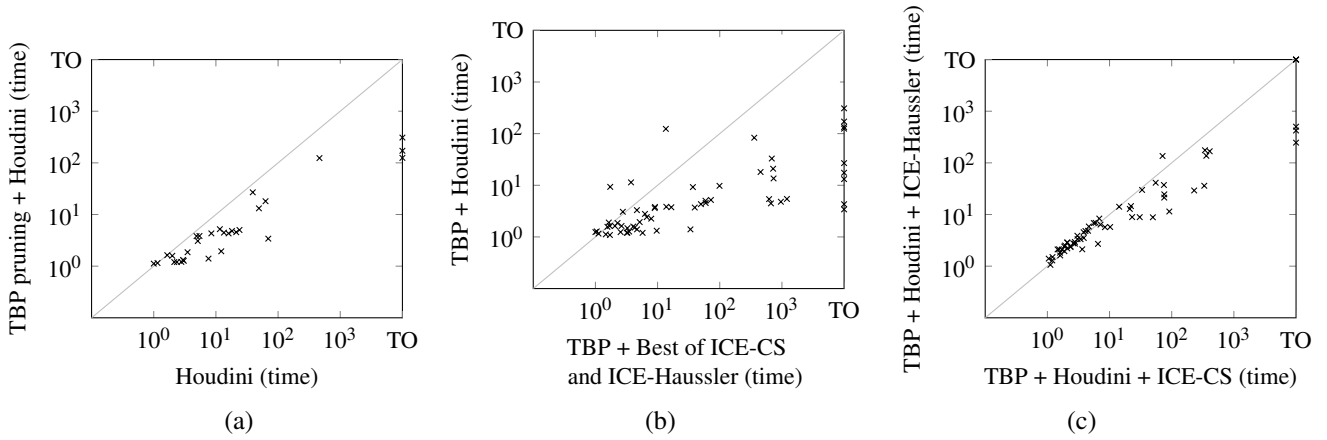


**Figure 3.** Scatter plot comparison. Time in s. TBP=Testing-based Pruning; TO=Timeout of 1800s


polants. In *CAV Berkeley*, volume 7358 of *LNCS*, pages 679–685. Springer, 2012.

[4] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

[5] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, pages 52–68, 2005.

[6] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pages 115–137, 2005.

[7] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson. Gpuverify: a verifier for GPU kernels. In *OOPSLA 2012*, pages 113–132. ACM, 2012.

[8] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 184–190, 2011.

[9] A. Bouajjani, C. Dragoi, C. Enea, A. Rezine, and M. Sighireanu. Invariant synthesis for programs manipulating lists with unbounded data. In *CAV 2010*, volume 6174 of *LNCS*, pages 72–88. Springer, 2010.

[10] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 578–589, 2011.

[11] A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI 2011*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[12] J. Brotherston, C. Fuhs, J. A. N. Pérez, and N. Gorogiannis. A decision procedure for satisfiability in separation logic with inductive predicates. In T. A. Henzinger and D. Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 25:1–25:10. ACM, 2014. ISBN 978-1-4503-2886-9.

[13] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.

[14] B. E. Chang and X. Rival. Relational inductive shape analysis. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 247–260, 2008.

[15] E. Y. Chen, S. Chen, S. Qadeer, and R. Wang. Securing multiparty online services via certification of symbolic transactions. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 833–849. IEEE Computer Society, 2015.

[16] Y.-F. Chen and B.-Y. Wang. Learning boolean functions incrementally. In *CAV 2012*, volume 7358 of *LNCS*, pages 55–70. Springer, 2012.

[17] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9): 1006–1036, Aug. 2012. ISSN 0167-6423.

[18] D. Chu, J. Jaffar, and M. Trinh. Automatic induction proofs of data-structures in imperative programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 457–466. ACM, 2015.

[19] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[20] E. Cohen, W. J. Paul, and S. Schmaltz. Theory of multi core hypervisor verification. In *SOFSEM 2013: Theory and Practice of Computer Science, 39th International Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 26-31, 2013. Pro-*

*ceedings*, volume 7741 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 2013.

[21] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV 2003*, volume 2725 of *LNCS*, pages 420–432. Springer, 2003.

[22] B. Cook, C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *Proceedings of the 22Nd International Conference on Concurrency Theory*, CONCUR'11, pages 235–249, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23216-9.

[23] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL 1977*, pages 238–252. ACM Press, 1977.

[24] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 105–118, 2011.

[25] L. M. de Moura and N. Bjørner. Efficient e-matching for SMT solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, pages 183–198, 2007.

[26] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[27] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[28] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 287–302, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-33056-9, 978-3-540-33056-1.

[29] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE 2000*, pages 449–458. ACM Press, 2000.

[30] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[31] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *FME 2001*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.

[32] R. W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Proceedings of a Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–31, Providence, 1967. American Mathematical Society.

[33] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Learning universally quantified invariants of linear data structures. In *CAV 2013*, volume 8044 of *LNCS*, pages 813–829. Springer, 2013.

[34] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *CAV 2014*, volume 8559 of *LNCS*, pages 69–87. Springer, 2014.

[35] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Quantified data automata for linear data structures: a register automaton model with applications to learning invariants of programs manipulating arrays and lists. *Formal Methods in System Design*, 47(1):120–157, 2015.

[36] P. Garg, P. Madhusudan, D. Neider, and D. Roth. Learning Invariants using Decision Trees and Implication Counterexamples. In *Proceedings of the 43nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, to appear.

[37] Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 306–320, 2009.

[38] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 235–246, 2008.

[39] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI 2008*, pages 281–292. ACM, 2008.

[40] A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In *CAV 2009*, volume 5643 of *LNCS*, pages 634–640. Springer, 2009.

[41] D. Haussler. Quantifying inductive bias: AI learning algorithms and valiant's learning framework. *Artif. Intell.*, 36(2):177–221, 1988.

[42] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 165–181. USENIX Association, 2014.

[43] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17, 2015.

[44] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[45] R. Iosif, A. Rogalewicz, and J. Simacek. The tree width of separation logic with recursive definitions. In *Proceedings of the 24th International Conference on Automated Deduction*, CADE'13, pages 21–38, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-38573-5.

[46] S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 756–772. Springer, 2013.

[47] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS 2006*, volume 3920 of *LNCS*, pages 459–473. Springer, 2006.

[48] R. Jhala and K. L. McMillan. Array abstractions from proofs. In *CAV 2007*, volume 4590 of *LNCS*, pages 193–206. Springer, 2007.

[49] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham. Property-directed inference of universal invariants or proving their absence. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 583–602, 2015.

[50] S. Kong, Y. Jung, C. David, B. Wang, and K. Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In *APLAS 2010*, volume 6461 of *LNCS*, pages 328–343. Springer, 2010.

[51] S. Krishna, C. Puhrsch, and T. Wies. Learning invariants using decision trees. *CoRR*, abs/1501.04725, 2015.

[52] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *CAV 2012*, volume 7358 of *LNCS*, pages 427–443. Springer, 2012.

[53] Q. L. Le, C. Gherghina, S. Qin, and W. Chin. Shape analysis via second-order bi-abduction. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 52–68, 2014.

[54] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR 2010*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[55] P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive proofs for inductive tree data-structures. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 123–136, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3.

[56] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying security invariants in expressos. In *ASPLOS 2013*, pages 293–304. ACM, 2013.

[57] K. L. McMillan. Interpolation and SAT-Based model checking. In *CAV 2003*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[58] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS 2008*, volume 4963 of *LNCS*, pages 413–427. Springer, 2008.

[59] J. A. Navarro Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 556–566, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8.

[60] D. Neider. *Applications of Automata Learning in Versification and Synthesis*. PhD thesis, RWTH Aachen University, April 2014.

[61] E. Pek, X. Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in C using separation logic. In *PLDI 2014*, page 46. ACM, 2014.

[62] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using SMT. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 773–789. Springer, 2013.

[63] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic with trees and data. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 711–728. Springer, 2014. ISBN 978-3-319-08866-2.

[64] R. Piskac, T. Wies, and D. Zufferey. Grasshopper - complete heap verification with mixed specifications. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 2014. ISBN 978-3-642-54861-1.

[65] X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI 2013*, pages 231–242. ACM, 2013.

[66] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002*, pages 55–74. IEEE Computer Society, 2002.

[67] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM. ISBN 1-58113-095-3.

[68] M. N. Seghir, A. Podelski, and T. Wies. Abstraction refinement for quantified array assertions. In *SAS 2009*, volume 5673 of *LNCS*, pages 3–18. Springer, 2009.

[69] R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *CAV 2014*, volume 8559 of *LNCS*, pages 88–105. Springer, 2014.

[70] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *CAV 2012*, volume 7358 of *LNCS*, pages 71–87. Springer, 2012.

[71] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP 2013*, volume 7792 of *LNCS*, pages 574–592. Springer, 2013.

[72] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *SAS 2013*, volume 7935 of *LNCS*, pages 388–411. Springer, 2013.

[73] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 99–110, 2010.

## A. Proof of Theorem 2

In preparation of the proof of Theorem 2, let us first state a simple property of $\tau$ and $\kappa$, which we exploit later.

**Lemma 1.** *Let $X, Y \subseteq \mathcal{P}$ be two sets of predicates. Then, $X \cap Y = \emptyset$ if and only if $\tau(X) \models \kappa(Y)$.*

*Proof.* To proof Lemma 1, we require the notation of what it means that a Boolean variable occurs in a conjunction. To this end, let $\mathcal{B} = \{b_1, \ldots, b_n\}$ be a set of Boolean variables and $\varphi = b_{i_1} \wedge \ldots \wedge b_{i_\ell}$ with $1 \leq \ell \leq n$ and $i_j \in \{1, \ldots, n\}$ for $j \in \{1, \ldots, \ell\}$ a conjunction over $\mathcal{B}$. We then write

$$b \in \varphi \text{ if and only if } b \in \{b_{i_1}, \ldots, b_{i_\ell}\}$$

(read "$b_i$ occurs in $\varphi$").

We can now prove Lemma 1. Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be a finite set of predicates and $X, Y \subseteq \mathcal{P}$ two sets. Moreover, let $\kappa(X) = (v_1, \ldots, v_n)$ and $\tau(Y) = \varphi$.

***From left to right:*** Let $X \cap Y = \emptyset$. By definition of $\tau$, $v_i = 1$ if and only if $p_i \notin X$ and, hence, $v_i = 1$ for all $p_i \in \mathcal{P} \setminus X$. By definition of $\kappa$, on the other hand, $b_i \in \varphi$ if and only if $p_i \in Y$. Moreover, since $X \cap Y = \emptyset$, we have $Y \subseteq \mathcal{P} \setminus X$. Hence, $v_i = 1$ if $b_i \in \varphi$ and, thus, $(v_1, \ldots, v_n) \models \varphi$ (i.e., $\kappa(X) \models \tau(Y)$).

***From right to left:*** Let $\tau(X) \models \kappa(Y)$ (i.e., $(v_1, \ldots, v_n) \models \varphi$). By definition of $\models$, this means that $b_i \in \varphi$ implies $v_i = 1$. Moreover, $b_i \in \varphi$ implies $p_i \notin X$ by definition of $\kappa$ and, hence, $p_i \in Y$ implies $p_i \notin X$ by definition of $\tau$. Hence, $X \cap Y = \emptyset$. □

To prove Theorem 2, let us briefly recall the definition of consistency of a set with a sample: a set $X \subseteq \mathcal{P}$, where $\mathcal{P}$ is a finite set of predicates, is said to be consistent with a sample $\mathfrak{S} = (AS, DS, DC)$ if

a) $X \cap T_+ = \emptyset$ for every $T_+ \in AS$;

b) $X \not\subseteq T_-$ for every $T_- \in DS$; and

c) if $X \subseteq U_1$, then $X \cap U_2 = \emptyset$ for every $(U_1, U_2) \in DC$.

Let us now bring this definition into an equivalent, yet more suitable form: a set $X \subseteq \mathcal{P}$ is consistent with a sample $\mathfrak{S} = (AS, DS, DC)$ if

a′) $X \cap T_+ = \emptyset$ for every $T_+ \in AS$;

b′) $X \cap (\mathcal{P} \setminus T_-) \neq \emptyset$ for every $T_- \in DS$; and

c′) if $X \cap (\mathcal{P} \setminus U_1) = \emptyset$, then $X \cap U_2 = \emptyset$ for every $(U_1, U_2) \in DC$.

We can now prove Theorem 2.

*Proof.* We show the direction from left to right. The reverse direction is analogous and, therefore, skipped.

Let $X \subseteq \mathcal{P}$ and assume that $X$ is consistent with $\mathfrak{S} = (AS, DS, DC)$. We now show that $\kappa(X)$ is consistent with $\tau(\mathfrak{S}) = (S_+, S_-, S_\Rightarrow)$ by showing

- $\vec{v} \models \kappa(X)$ for all $\vec{v} \in S_+$;
- $\vec{v} \not\models \kappa(X)$ for all $\vec{v} \in S_-$; and
- if $\vec{v}_1 \models \kappa(X)$, then $\vec{v}_2 \models \kappa(X)$ for all $(\vec{v}_1, \vec{v}_2) \in S_\Rightarrow$.

Let us investigate positive examples, negative examples, and implications individually.

***Positive examples:*** Pick some element $\vec{v} \in S_+$, say $\vec{v} = \tau(T_+)$ where $T_+ \in AS$ (recall the definition of $\tau(\mathfrak{S})$). Since $X$ is consistent with $\mathfrak{S}$, we know that $X \cap T_+ = \emptyset$. By Lemma 1, we immediately obtain $\vec{v} = \tau(T_+) \models \kappa(X)$.

***Negative examples:*** Pick some element $\vec{v} \in S_-$, say $\vec{v} = \tau(\mathcal{P} \setminus T_-)$ where $T_- \in DS$. Since $X$ is consistent with $\mathfrak{S}$, we know that $X \cap (\mathcal{P} \setminus T_-) \neq \emptyset$. By Lemma 1, we immediately obtain $\vec{v} = \tau(\mathcal{P} \setminus T_-) \not\models \kappa(X)$.

***Implications:*** Pick some implication $(\vec{v}_1, \vec{v}_2) \in S_\Rightarrow$, say $\vec{v}_1 = \tau(\mathcal{P} \setminus U_1)$ and $\vec{v}_2 = \tau(U_2)$ where $(U_1, U_2) \in DC$. Since $X$ is consistent with $\mathfrak{S}$, we know that $X \cap (\mathcal{P} \setminus U_1) = \emptyset$ implies $X \cap U_2 = \emptyset$. By Lemma 1, we immediately obtain that if $\vec{v}_1 = \tau(\mathcal{P} \setminus U_1) \models \kappa(X)$, then $\vec{v}_2 = \tau(U_2) \models \kappa(X)$.

Since $\kappa(X)$ satisfies all consistency constraints, $\kappa(X)$ is consistent with $\tau(\mathfrak{S})$. □