Learning Invariants using Decision Trees and Implication Counterexamples

Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth

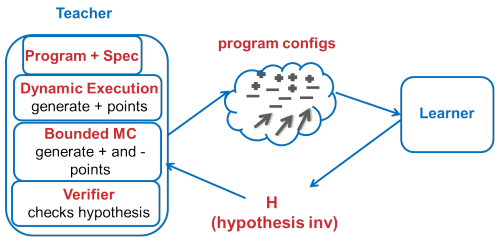
University of Illinois at Urbana-Champaign

Abstract. Inductive invariants can be robustly synthesized using a learning model where the teacher is a program verifier who instructs the learner through concrete program configurations, classified as positive, negative, and implications. We propose the first learning algorithms in this model with implication counter-examples that are based on scalable machine learning techniques. In particular, we extend decision tree learning algorithms, building new scalable and heuristic ways to construct small decision trees using statistical measures that account for implication counterexamples. We implement the learners and an appropriate teacher, and show that they are scalable, efficient and convergent in synthesizing adequate inductive invariants in a suite of more than 50 programs.

1 Introduction

Automatically synthesizing invariants, in the form of inductive pre / post conditions and loop invariants, is a challenging problem that lies at the heart of automated program verification. If an adequate inductive invariant is found or given by the user, the problem of checking whether the program satisfies the specification can be reduced to logical validity of verification conditions, which is increasingly tractable with the advances in automated logic solvers.

In recent years, the *black-box* or *learning* approach to finding invariants has gained popularity [45, 46, 44, 22, 23], in contrast to white-box approaches such as interpolants, methods using Farkas' lemma, IC3, etc. [33, 28, 14, 25, 26, 11]. In this data-driven approach, we split the synthesizer of invariants into two parts (see figure to the right). One component is a *teacher*, which is essentially a program verifier that can verify the program using a conjectured invariant and generates counter-examples; it may also have other ways of generating configurations that must or must not be in the invariant (e.g., dynamic execution engines, bounded model-checking engines, etc.). The other component is a *learner*, which learns from counter-examples given by the teacher to synthesize the invariant. In each round, the learner proposes an invariant hypothesis H, and the teacher checks if the hypothesis is adequate to verify the program against



the specification; if not, it returns concrete program configurations that are used in the next round by the learner to refine the conjecture. The most important feature of this framework is that the learner is completely agnostic of the program and the specification (and hence the semantics of the programming language, memory model, etc.). The learner is simply constrained to learn some formula (or predicate) that satisfies the sample configurations given by the teacher.

ICE Learning Model The simplest way for the teacher to refute an invariant is to give positive and negative program configurations, S^+ and S^-, constraining the learner to find a predicate that includes S^+ and excludes S^-. However, this is not always possible. In a recent paper, Garg et al. [23] note that if the learner gives a hypothesis that covers all states known to be positive by the teacher and excludes all states known to be negative by the teacher, but yet is not *inductive*, then the teacher is stuck and cannot give any positive or negative counter-example to refute the hypothesis.

Garg et al. [23] define a new learning model, which they call *ICE* (for implication counter-examples) that allows the teacher to give counter-examples of the form (x, y), where both x and y are program configurations, with the constraint that the learner must propose a predicate such that if it includes x, then it includes y as well. These implication counter-examples can be used to refute non-inductive invariants: if H is not inductive, then the teacher can find a configuration x satisfying H such that x evolves to y in the program but y is not satisfied by H. This learning model forms a robust paradigm for learning invariants, including loop invariants, multiple loop invariants, and nested loop invariants in programs [23]—the teacher can be both *honest* (never give an example classification that precludes an invariant) and make *progress* (always be able to refute an invariant that is not inductive or adequate). This is in sharp contrast to learning only from positive and negative examples, where the teacher is forced to be dishonest (as it does not know an invariant) to make progress.

Machine Learning for Finding Invariants One of the biggest advantages of the black-box learning paradigm is the possible usage of scalable *machine learning techniques* to synthesize invariants. The learner, being completely agnostic to the program (its programming language, semantics, etc.), can be seen as a machine learning algorithm that learns a Boolean classifier of configurations. Machine learning algorithms can be trained to learn functions that belong to various Boolean functions, such as k-CNF/k-DNF, Linear Threshold Functions, Decisions Trees, etc., and some algorithms have already been used in invariant generation [47]. However standard machine learning algorithms for classification are trained on given sets of positive and negative examples (but do not handle implications), and hence do not help in building robust learning platforms for invariant generation.

In this paper, we build the first true machine learning algorithms for robust invariant generation. We adapt and extend classical scalable machine learning algorithms for constructing decision trees (which can express any Boolean function) to scalable ICE learning algorithms. We show that this results in scalable and fast algorithms for synthesizing invariants.

Decision trees over a set of attributes provide a universal representation of a Boolean function defined over this set of numerical and categorical attributes. Internal nodes in decision trees are decision variables that split over a value of a single attribute, and the leaves are labeled with classification labels (positive or negative in our setting).

Our starting point is the well-known decision tree learning algorithms of Quinlan [39, 40, 36] that work by constructing the tree top-down from positive and negative samples. These are extremely scalable algorithms as they choose heuristically the best attribute that classifies the sample at each stage of the tree based on statistical measures, and do not backtrack nor look ahead. One of the well-known ways to pick these attributes is based on a notion called *information gain*, which is in turn based on a statistical measure using Shannon's entropy function [43, 40, 36]. The inductive bias in these learning algorithms is roughly to compute the *smallest* decision tree that is consistent with the sample— a bias that again suits our setting well, as we would like to construct the smallest invariant formula amongst the large number of invariants that may exist. Machine learning algorithms, including the decision tree learning algorithm, often do not produce concepts that are fully consistent with the given sample— this is done on purpose to avoid over-fitting to the training set, under the assumption that, once learned, the hypothesis will be evaluated on new, previously unseen data. We remove these aspects from the decision tree algorithm (which includes, for example, pruning to produce smaller trees at the cost of inconsistency) as we aim at identifying a hypothesis that is *correct* rather than one that only *approximates* the target hypothesis.

Our first technical contribution is a generic top-down decision tree algorithm that works on samples with implications. This algorithm constructs a tree top-down, classifying end-points of implications in a way that reduces the sizes of trees and manages always to create a tree that is consistent with the sample. Our second technical contribution is a study of various natural measures for learning decisions tree in the presence of positive, negative examples, and implications. We do this by developing several novel "information gain" measures that are used to determine the best attribute to split on given the current collection of examples and implications. The first naive metric is to simply ignore implications when choosing attributes— however, ignoring implications entirely seems to result in non-robust invariant generation, where even on simple examples the learning diverges. The second metric that we propose is a natural extension of the entropy measure to account for implications, where we consider the unclassified examples as probabilistically classified, under the constraints imposed by the implications. A third proposal stems from the observation that in most of the generated invariants, the end-points of implications are mostly classified either both positively or both negatively. Consequently, the third measure we propose is one that uses the classical entropy for positive and negative points, but imposes a *weighted penalty* based on the number of implications that are *cut* by the considered attribute (i.e., where one end-point satisfies the attribute and the other does not), and which do not connect a predominantly negative set to a

predominantly positive set. We also tried several other intuitive statistical ways to define heuristic measures to find the best attributes, but the two described above gave the best results.

We implement our ICE decision tree algorithms (i.e., the generic ICE learning algorithm with the various statistical measures for choosing attributes) and build teachers to work with these learners to guide them towards learning invariants. We perform extensive experiments that show that the decision tree learners we build are competitive, and superior in performance and convergence to the constraint-solving based ICE learner presented in [23]. We use our tool to find invariants in around 50 programs, and the new learning techniques work extremely well; surprisingly, they converge on all examples. We also perform experiments that show, in general, how scalable the decision tree learning is in learning from large sets of points compared with techniques based on constraint solving.

We believe that this work breaks new ground in adapting machine learning techniques to invariant synthesis, giving the first scalable and robust ICE machine-learning algorithms for synthesizing invariants.

Related Work Algorithms for invariant synthesis can be broadly categorized as white-box techniques and black-box techniques. Prominent examples for white-box techniques include abstract interpretation [16], interpolation [33, 28], and IC3 [11]. Abstract interpretation is predominately used for synthesizing invariants over convex domains [35, 17, 6, 27], but there also exist applications to non-convex domains [19, 42] and even non-lattice domains [21]. Template-based approaches for synthesizing invariants using constraint solvers have been also explored in a white-box setting [14, 25, 26]. On the other hand, a prominent early example of black-box techniques for synthesizing invariants is Daikon [18], which uses conjunctive Boolean learning to find *likely* invariants from test runs. Active learning in the context of verification was introduced by Cobleight et al. [13], followed by applications of Angluin's L^* [4] to verification problems such as finding rely-guarantee contracts [3] and stateful interface specifications for programs [2], verifying CTL properties [51], and Regular Model Checking [38]. Houdini [20], which learns conjunctive Boolean invariants, is another prominent algorithm. Work on liquid types [30] uses a similar algorithm for inferring refinement types for program expressions.

Recently, learning has gained renewed interest in the context of program verification, particularly for synthesizing loop invariants [47, 45, 46, 48, 22]. However, Garg et al. [23] argue that merely learning from positive and negative examples for synthesizing invariants is inherently non-robust and introduce ICE-learning, which extends the classical learning setting with implications. Implication counter-examples were also identified by Sharma et al [46], but the learners proposed did not handle them in any way. Examples of algorithms using ICE-learning have been subsequently proposed for learning invariants over octogonal domains and universally quantified invariants over linear data structures [22], Regular Model Checking [37], and a general framework for generating invariants based on randomized search [44]. Some generalizations of Houdini [50, 24] can also be seen as ICE-learning algorithms. In *program* or *expression* synthesis, a popular approach

to synthesis is using counter-example guided inductive synthesis (CEGIS), which is also a black-box learning paradigm [1, 49] like ICE, and is gaining traction aided by explicit enumeration, symbolic constraint-solving and stochastic search algorithms.

Machine learning algorithms (see [36] for an overview) are often used in practical learning scenarios due to their high scalability. Algorithms for learning linear classifiers include winnow [31], perceptron [41], and support vector machines [15]. Since invariants in our current work are arbitrary Boolean functions, our learner is build on decision tree algorithms such as ID3 [39], C4.5 [40] and C5.0. Apart from these classification algorithms, algorithms for learning more complicated structured objects using structured prediction [5] have also become popular recently.

2 Background: Learning Decision Trees from Positive and Negative Examples

Our algorithm for learning invariants builds on the classical recursive algorithm to build *decision trees* proposed by Quinlan (we refer the interested reader to standard texts on learning for more information on decision tree learning [36]). The reader is encouraged to think of decision trees as Boolean combinations of formulae of the form $a_i \leq c$, where a_i is drawn from a fixed set of *numerical attributes* A (which assign a numerical value to each sample) and c is a constant or of the form b_i, where b_i is drawn from a fixed set of *Boolean attributes* (which assign a truth value to each sample). When performing invariant learning, we will fix a set of attributes typically as certain arithmetic combinations of integer variables (for example, octagonal combinations of variables or certain linear combinations of variables with bounded co-efficients). Boolean attributes are useful for other non-numerical constraints (are x and y aliased, does x point to *nil*, etc.). Consequently, the learner would learn the thresholds (the values for c in $a_i \leq c$) and the Boolean combination of the resulting predicates, including arbitrary conjunctions of disjunctions as a proposal for the invariant.

Quinlan's algorithm, sketched in pseudo code as Algorithm 1, builds the tree top-down (without backtracking), choosing the best attribute at each stage using an information theoretic measure called *information gain*. It has been implemented by the ID3, C4.0, and C5.0 algorithms [39, 40, 36].

The crucial aspect of the extremely scalable decision tree learning algorithms is that they choose the attribute for the current sample in some heuristic manner, and never back-track (or look forward) to optimize the size of the decision tree. The prominent technique for choosing attributes is based on a statistical property, called *information gain*, to measure how well each attribute classifies the examples at any stage of the algorithm. This measure is typically defined using a notion called *Shannon entropy* [43], which, intuitively, captures the impurity of a sample. The entropy of a sample S with p positive samples and n negative samples is a value between 0 and 1, defined to be

$$\mathit{Entropy}(S) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}.$$

input : A sample $S = \langle S^+, S^- \rangle$ and *Attributes*

1 Return ID3 ($\langle S^+, S^- \rangle$, *Attributes*).

Proc ID3 *(Examples* $= \langle Pos, Neg \rangle$*, Attributes)*

2 Create a root node of the tree.

3 **if** *all examples are positive or all are negative* **then**

4 Return the single node tree Root with label + or −, respectively.

else

5 Select an attribute a (and a threshold c for a if a is numerical) that (heuristically) *best* classifies *Examples*.

6 Label the root of the tree with this attribute (and threshold).

7 Divide *Examples* into two sets: $Examples_a$ that satisfy the predicate defined by attribute (and threshold), and $Examples_{\neg a}$ that do not.

8 Return tree with root and left tree ID3 ($Examples_a$, *Attributes*) and right subtree ID3 ($Examples_{\neg a}$, *Attributes*) ;

end

Algorithm 1: The basic inductive decision tree construction algorithm underlying ID3, C4.0, and C5.0

Intuitively, if the sample contains only positive (or only negative) points (i.e., if $p = 0$ or $n = 0$), then its entropy is 0, while if the sample had roughly an equal number of positive and negative points, then its entropy is close to 1.

When evaluating an attribute a (and threshold) on a sample S, splitting it into S_a and $S_{\neg a}$ (points satisfying the attribute and points that do not), one computes the information gain of that attribute (w.r.t. the chosen threshold): the information gain is the difference between the entropy of S and the sum of the entropies of S_a and $S_{\neg a}$ weighted by the number of points in the respective samples. For numerical attributes, the thresholds also need to be synthesized; in the case of information gain, however, it turns out that the best thresholds is always at some value occurring in the points in the sample. The algorithm chooses the attribute that results in the largest information gain.

The above heuristic for greedily picking the attribute that works best at each level has been shown to work very well in large and a wide variety of machine learning applications [40, 36]. When decision tree learning is used in machine learning contexts, there are other important aspects: (a) the learning is achieved using a small portion of the available sample, so that the tree learned can be evaluated for accuracy against the rest of the sample, and (b) there is a *pruning* procedure that tries to generalize and reduce the size of the tree so that the tree does not overfit the sample. When using decision tree learning for synthesizing invariants, we prefer to use all the samples as we anyway place the passive learning algorithm in a larger context by building a teacher which is a verification oracle. Also, we completely avoid the pruning phase since pruning often produces trees that are (mildly) inconsistent with the sample; since we cannot tolerate any inconsistent trees, we prefer to avoid this (incorporating pruning in a meaningful and useful way in our setting is an interesting future direction).

In our setting, we assume that all integer variables mentioned in the program occur explicitly as numerical attributes; hence, it turns out that *any* sample of mixed positive and negative points can be split (potentially using the same

attribute repeatedly with different thresholds) and eventually separated into purely positive and purely negative points (in the worst case, each point is separated into its own leaf). Consequently, we are guaranteed to always obtain some decision tree that is consistent with any input sample.

3 A Generic Decision Tree Learning Algorithm in the Presence of Implications

In this section, we present the skeleton of our new decision tree learning algorithm for samples containing implication examples in addition to positive and negative examples. We present this algorithm at the level of Algorithm 1, excluding the details of *how* the best attribute at each stage is chosen. In Section 4, we articulate several different natural ways of choosing the best attribute, and evaluate them in experiments.

Our goal in this section is to build a top-down construction of a decision tree for an *ICE sample*, such that the tree is guaranteed to be consistent with respect to the sample; an ICE sample is a tuple $S = (S^+, S^-, S^{\Rightarrow})$ consisting of a finite set S^+ of positive points, a finite set S^- of negative points, and a finite set $S^{\Rightarrow}$ of pairs of points corresponding to the implications. The algorithm is an extension of the classical decision tree algorithm presented in Section 2 and, hence, will automatically be consistent with positive and negative samples. The main hurdle we need to cross is to construct a tree consistent with the implications. Note that the pairs of points corresponding to implications do not have a classification, and it is the learner's task to come up with a classification in a manner consistent with the implication constraints. As part of the design, we would like the learner to not classify the points a priori in any way, but classify these points in a way that leads to a smaller concept (or tree).

Our algorithm, shown in pseudo code as Algorithm 2, works as follows. First, given an ICE sample $\langle S^+, S^-, S^{\Rightarrow}\rangle$ and a set of attributes, we store $S^{\Rightarrow}$ in a global variable *Impl* and create a set *Unclass* of unclassified points as the end-points of the implication samples. We also create a global table G that holds the partial classification of all the unclassified points (initially empty). We then call our recursive decision tree constructor with the sample $\langle S^+, S^-, \mathit{Unclass}\rangle$.

Receiving a sample $\langle \mathit{Pos}, \mathit{Neg}, \mathit{Unclass}\rangle$ of positive, negative, and unclassified examples, our algorithm chooses the best attribute that divides this sample, say a, and recurses on the two resulting samples $\mathit{Examples}_a$ and $\mathit{Examples}_{\neg a}$. Unlike the classical learning algorithm, we do not recurse *independently* on the two sets—rather we recurse first on $\mathit{Examples}_a$, which will, while constructing the left subtree, make classification decisions on some of the unclassified points, which in turn will affect the construction of the right subtree for $\mathit{Examples}_{\neg a}$ (see the `else` clause in Algorithm 2). The new classifications that are decided by the algorithm are stored and communicated using the global variable G.

Whenever Algorithm 2 reaches a node where the current sample has only positive points and implication end-points that are either classified positively or unclassified yet, then the algorithm will, naturally, decide to mark *all* remaining

input : An ICE sample $S = \langle S^+, S^-, S^{\Rightarrow} \rangle$ and *Attributes*

Initialize global *Impl* to $S^{\Rightarrow}$.
Initialize G, a partial valuation of end-points of implications in $S^{\Rightarrow}$, to be empty.
Let *Unclass* be the set of all end-points of implications in $S^{\Rightarrow}$.
Take implication closure of G with respect to the positive and negative classifications in S^+ and S^-.
Return `DecTreeICE` ($\langle S^+, S^-,$ *Unclass*$\rangle$, *Attributes*).

Proc `DecTreeICE` *(Examples* $= \langle$*Pos*, *Neg*, *Unclass*$\rangle$*, Attributes)*

    Move all points from *Unclass* that are positively, respectively negatively, classified in G to *Pos*, respectively *Neg*.
    Create a root node of the tree.
    if $Neg = \emptyset$ **then**
        Mark all elements in *Unclass* as positive in G.
        Take the implication closure of G w.r.t. *Impl*.
        Return the single node tree Root, with label $+$.
    else if $Pos = \emptyset$ **then**
        Mark all elements in *Unclass* as negative in G.
        Take the implication closure of G wrt *Impl*.
        Return the single node tree Root, with label $-$.
    else
        Select an attribute a (and a threshold c for a if a is numerical) that (heuristically) *best* classifies *Examples* and *Impl*.
        Label the root of the tree with this attribute a (and threshold c).
        Divide *Examples* into two sets: $Examples_a$ that satisfy the predicate defined by the attribute (and threshold) and $Examples_{\neg a}$ that do not.
        $T_1 = DecTreeICE(Examples_a, Attributes)$ (may update G).
        $T_2 = DecTreeICE(Examples_{\neg a}, Attributes)$ (may update G).
        Return tree with root, left tree T_1, and right tree T_2.
    end

Algorithm 2: The basic decision tree construction algorithm for ICE samples

unclassified points positive, and declare the current node to be a leaf of the tree (see first conditional in the algorithm). Moreover, it marks in G all the unclassified end-points of implications in *Unclass* as positive and propagates this constraint across implications (taking the implication closure of G with respect to the global set *Impl* of implications). For instance, if (x, y) is an implication pair, both x and y are yet unclassified, and the algorithm decides to classify x as positive, it propagates this constraint by making y also positive in G; similarly, if the algorithm decided to classify y as negative, then it would mark x also as negative in G. Deciding to classify x as negative or y as positive places no restrictions on the other point, of course.

The next theorem states that Algorithm 2 always terminates and constructs a decision tree that is consistent with the ICE sample. The reason is that classifying end points of implications in a leaf node as uniformly positive or uniformly negative and taking the implication closure will never violate an implication (note that the implication could start from a leaf to a node that is yet to be explored). A complete proof of the theorem can be found in Appendix B.

Theorem 1. Algorithm 2, independent of how the attributes are chosen to split a sample, always terminates and produces a decision tree that is consistent with the input ICE sample.

The running time of Algorithm 2 depends, of course, on the time taken to choose the best attribute in each recursive call. Apart from this, however, the algorithm is linear in the size of the sample: in each round, the sample set is divided into two parts and recursed on, and the total updates to G and the sum of the implication closures on G can be done in time linear in the number of implications. We explore several different ways to choose the best attribute at each stage in the next section, all of which are linear or quadratic on the sample.

4 Choosing Attributes in the Presence of Implications

Algorithm 2 ensures that the resulting decision tree is consistent with the given sample, irrespective of the exact mechanism used to determine the attributes to split and their thresholds. As a consequence, the original split heuristic based on information gain (see Section 2), which is unaware of implications, might simply be employed. However, since our algorithm propagates the classification of data points once a leaf node is reached, just ignoring implications can easily lead to splits that are good at the time of the split but later turn into bad ones since the classification of points has changed (Appendix A illustrates such a situation). In fact, our experiments show that a learner which ignores implications when choosing an attribute often learns larger decision trees or even diverges.

We next propose two methods that take implications into account while choosing the attribute to split. We observed that these methods yield faster convergence when used in an invariant synthesis setting.

Computing Information Gain for an ICE Sample: The entropy of a set of examples is a function of the discrete probability distribution of the classification of a point drawn randomly from the examples. In a classical sample that only has points labeled positive or negative, one could count the fraction of positive (or negative) points in the set to compute these probabilities. However, an estimation of these probabilities becomes non-trivial in the presence of unclassified points that can be probabilistically classified as either positive or negative. Moreover, in an ICE sample, the classification of these points is not independent anymore as the classification for the points need to satisfy the implication constraints. Given a set of examples with implications and unclassified points, we will first estimate the probability distribution of the classification of a random point drawn from these examples, taking into account the implication constraints, and then use it for computing the entropy. We will use this new entropy to compute the information gain while choosing the attribute for the split.

Given $S = \langle \mathit{Pos}, \mathit{Neg}, \mathit{Unclass} \rangle$, and a set of implications Impl, let Impl_S be the set of implications projected onto S such that both the antecedent and consequent points in the implication are unclassified (i.e., $\mathit{Impl}_S = \{(x_1, x_2) \in \mathit{Impl} \mid x_1, x_2 \in \mathit{Unclass}\}$). For the purpose of entropy computation, we will

assume that there is no point in the examples that is common to more than one implication. This is a safe assumption if the space enclosing all the points is much larger than the number of points. Also, let *Unclass'* $\subseteq$ *Unclass* be the set of unclassified points in the sample that are not part of any implication in $Impl_S$ (for example x_1 where $(x_1, x_2) \in$ *Impl and* $x_2 \in$ *Pos*). Note that points in *Unclass'* can be classified as positive or negative, independent of the classification of other points.

Let $Pr(x = a)$ be the probability of a point $x \in S$ being classified as a where a is positive $(+)$ or negative $(-)$. Note that $Pr(x = +)$ is one for $x \in$ *Pos* and zero for $x \in$ *Neg*. Also, let $Pr(S, a)$ be the probability of a point drawn randomly from S being classified as a. Then,

$$Pr(S,+) \;=\; \frac{1}{|S|}\sum_{x\in S} Pr(x=+) \;=\; \frac{1}{|S|}\Big(\sum_{x\in Pos\cup Neg\cup Unclass'} Pr(x=+) \;+\; \sum_{(x_1,x_2)\in Impl_S} Pr(x_1=+) \;+\; Pr(x_2=+)\Big) \quad (1)$$

We assume that unclassified points $x_U \in$ *Unclass'* and points $x_1 \in S$ where $(x_1, x_2) \in Impl_S$ are classified independently and probabilistically in accordance with the distribution of points in the set S. In other words, we recursively assign $Pr(x_U = +) = Pr(x_1 = +) = Pr(S, +)$. However, the classification of x_2 depends on the classification of x_1 as the implication constraint between them needs to be satisfied. Using conditional probabilities we obtain, $Pr(x_2 = +) = Pr(x_2 = + \mid x_1 = +) \cdot Pr(x_1 = +) + Pr(x_2 = + \mid x_1 = -) \cdot Pr(x_1 = -)$. From the implication constraint between x_1 and x_2, $Pr(x_2 = + \mid x_1 = +) = 1$. However, $Pr(x_2 = + \mid x_1 = -) = Pr(S, +)$ since x_2 can be classified independently when x_1 is classified negative. Plugging in these values for probabilities in Equation 1 and using $p = |Pos|, n = |Neg|, i = |Impl|, u' = |$*Unclass'*$|$ *and* $|S| = p + n + 2i + u'$, $Pr(S, +)$ is the positive solution of the following quadratic equation:

$$ix^2 + (p + n - i)x - p = 0$$

As a sanity check, note that $Pr(S, +) = \frac{p}{p+n}$, if there are no implications in the sample set (i.e., $i = 0$). Also, $Pr(S, +) = 1$ if $n = 0$ and $Pr(S, +) = 0$ if $p = 0$ (i.e., when the set S has no negative or positive points). Once we have computed $Pr(S, +)$, the entropy of S can be computed in the standard way as

$$Entropy(S) = -Pr(S,+) \cdot \log_2\ Pr(S,+) - Pr(S,-) \cdot \log_2\ Pr(S,-)$$

where $Pr(S, -) = \big(1 - Pr(S, +)\big)$. Now, we plug this new entropy in the information gain and obtain a gain measure that explicitly takes implications into account.

Penalizing Cutting Implications In order to better understand how to deal with implications, we analyzed classifiers learned by other ICE-learning algorithms for invariant synthesis, such as the randomized search of [44] and the constraint solver-based ICE learner of [23]. This analysis showed that the classifiers finally learned (and also those conjectured during the learning) almost always classify the antecedent and consequents of implications equally (either both positive or both negative).

The fact that successful ICE learners almost always classify antecedents and consequents of implications equally suggests that our decision tree learner should

avoid to "cut" implications. Formally, we say that an implication $(p, q) \in \mathit{Impl}$ is *cut* by the samples S_a and $S_{\neg a}$ if $p \in S_a$ and $q \in S_{\neg a}$, or $p \in S_{\neg a}$ and $q \in S_a$;[1] in this case, we also say that the split of S into S_a and $S_{\neg a}$ cuts the implication.

A straightforward approach to discourage cutting implications builds on top of the original information gain and imposes a penalty for every implication that is cut. This idea gives rise to the *penalized information gain* that we define by

$$\mathit{Gain}_{pen}(S, S_a, S_{\neg a}) = \mathit{Gain}(S, S_a, S_{\neg a}) - \mathit{Penalty}(S_a, S_{\neg a}, \mathit{Impl})$$

where $S_a, S_{\neg a}$ is the split of the sample S, $\mathit{Gain}(S, S_a, S_{\neg a})$ is the original information gain based on Shannon's entropy, and $\mathit{Penalty}(S_a, S_{\neg a}, \mathit{Impl})$ is a total penalty function that we assume to be monotonically increasing with the number of implications cut (we make this precise shortly). Note that this new information gain does not prevent the cutting of implications (if this is necessary) but favors not to cut them.

However, not every cut implication poses a problem: implications whose antecedents are classified negatively and whose consequents are classified positively are safe to cut (as this helps creating more pure samples), and we do not want to penalize cutting those. Since we do not know the classifications of unclassified points when choosing an attribute, we penalize an implication depending on how "likely" it is an implication of this type (i.e., we assign no penalty if the sample containing the antecedent is predominantly negative and the one containing the consequent is predominantly positive). More precisely, given the samples S_a and $S_{\neg a}$, we define the penalty function $\mathit{Penalty}(S_a, S_{\neg a}, \mathit{Impl})$ by

$$\Big(\sum_{\substack{(x,y)\in \mathit{Impl}\\ x\in S_a, y\in S_{\neg a}}} 1 - f(S_a, S_{\neg a})\Big) + \Big(\sum_{\substack{(x,y)\in \mathit{Impl}\\ x\in S_{\neg a}, y\in S_a}} 1 - f(S_{\neg a}, S_a)\Big),$$

where for two samples $S_1 = \langle \mathit{Pos}_1, \mathit{Neg}_1, \mathit{Unclass}_1\rangle$, $S_2 = \langle \mathit{Pos}_2, \mathit{Neg}_2, \mathit{Unclass}_2\rangle$

$$f(S_1, S_2) = \frac{|\mathit{Neg}_1|}{|\mathit{Pos}_1| + |\mathit{Neg}_1|} \cdot \frac{|\mathit{Pos}_2|}{|\mathit{Pos}_2| + |\mathit{Neg}_2|}$$

is the relative frequency of the negative points in S_1 and the positive points in S_2 (which can be interpreted as likelihood of an implication pointing from S_1 to S_2 being safe).

5 Experiments and Evaluation

To assess the performance of our decision tree ICE learner, we implemented a prototype of Algorithm 2 as an invariant synthesis tool in Boogie [7] and benchmarked it to other verification algorithms. Our experimental setup is as follows.

Learner: We implemented Algorithm 2 (with the attribute selection methods described in Section 4) on top of the freely available version of the C5.0 algorithm (Release 2.10) [40]. Since we rely on learning without classification errors, we disabled all of C5.0's optimizations, such as pruning, boosting, etc.

[1] Given a sample $S = \langle \mathit{Pos}, \mathit{Neg}, \mathit{Unclass}\rangle$, we write $x \in S$ as a shorthand notation for $x \in \mathit{Pos} \cup \mathit{Neg} \cup \mathit{Unclass}$.

Teacher: We implemented a teacher in Boogie, which translates an input program into verification conditions and prototypes of functions that need to be synthesized. To learn invariants over octagonal predicates (of the form $\pm x \pm y \leq c$), we add auxiliary attributes of the form $\pm x \pm y$ for all combinations of variables x, y in the program (note that the learner learns the thresholds c as well as the Boolean combination of these predicates).

The actual learning starts with an empty ICE sample. Whenever the learner proposes a conjecture, the teacher checks whether it satisfies the verification conditions. If this is not the case, she derives a counterexample—either a spurious pos/neg data point or an implication—and returns it to the learner. The teacher favors to return pos/neg data points, and returns implications only if no spurious pos/neg data points can be found (we found that this performed best for our learners). Since loop invariants usually do not involve large constants, we employed the following search strategy: when searching for counterexamples, we successively bound the absolute values of the variables to 2, 5, and 10 and successively proceed to the next larger bound if no counterexample within the current bound exists; if no counterexample within any of these bounds exists, we fall back to the most general case and do not impose any bound.

Experimental Setup: We conducted all experiments on a Core i5 CPU with 6 GiB of RAM running Windows 7 using a 60 s timeout limit.

Scalability Benchmark: We begin with a benchmark demonstrating the scalability of the decision tree ICE learner compared to other learning techniques, namely the constraint solver-based ICE learner of [23] and the learner based on computational geometry from [46]. This benchmark is mainly intended for readers interested in machine learning techniques, independent of their application to invariant synthesis. Our benchmark comprises three sample suits, each consisting of randomly drawn samples containing between 50 and 50 000 data points each. The suits are classified by the formulas $\varphi_1 = x_1 \leq -1 \vee x_2 \geq 1$, $\varphi_2 = x_1 - x_2 \geq 2$, and $\varphi_3 = 0 \leq x_0 \wedge 0 \leq x_1 \wedge 0 \leq x_2 \wedge x_3 \neq 1 \wedge x_4 \neq 1 \wedge x_5 \neq 1$, respectively. The results of this benchmark are shown in Figure 1.

The decision tree-learner is on average one order of magnitude faster than the other algorithms and can handle samples up to 50 000 data points. Our benchmarks demonstrate that the decision tree-based ICE learner scales much better than other learners described in literature, which strongly suggests that machine-learning based tools may be very effective.

Invariant Generation: We evaluate the two configurations of the decision-tree based learner discussed in Section 4 in the context of invariant synthesis and compare them to the interpolation based Impact algorithm [34] implemented in CPAchecker [9] and the constraint based learning algorithm proposed in [23]. The experimental results are tabulated in Table 1. The first set of results correspond to the Impact algorithm [34] (called CPAchecker); next we tabulate results for the black-box learners including the constraint-solver based learner from [23] (called ICE-CS), the decision tree learner that computes the information gain that accounts for implications while deciding on the attribute to split (called ICE-DT-entropy), and the decision-tree learner that penalizes splits that cut

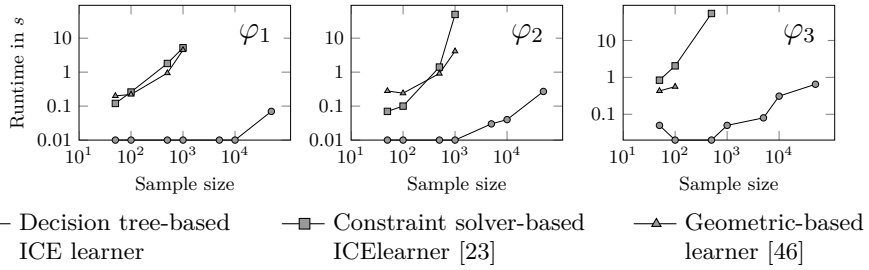


Fig. 1. Results of the scalability benchmark

implications (called ICE-DT-penalty). Note that the ICE-CS learner uses a slightly different teacher which does not bound constants in counter-examples as the learner in ICE-CS itself searches for invariants with smaller constants [23]. For each of the black-box learner we provide details about the composition of the final sample, in terms of the number of positive, negative and implication counter-examples that were required to learn an adequate invariant, the number of rounds to converge and the final time in seconds that was taken to learn an adequate invariant.

We evaluate the invariant synthesis tools on several programs taken from previous literature [23, 26][2]. Being white-box, CPAchecker is not precise for programs with arrays and pointers; also for several programs over numerical variables that have complex disjunctive assertions, it reports an error even when these programs are safe (reported as $\times$ in Table 1). Table 1 also records the time taken by various black-box learners aggregated over programs on which ICE-CS does not time out. Our decision tree based learners are $\sim$5x times faster than the constraint solver based learner ICE-CS on the benchmark suite. This is impressive given that ICE-CS in itself is quite a fast learner and has been compared in [23] to various tools including Invgen [26], Houdini [50] and the Impact algorithm [34] from CPAchecker [9]. From the table, we make the following key observations:

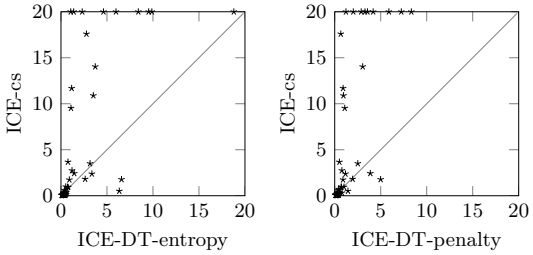


Fig. 2. Runtime comparison (in s) of ICE-DT-penalty and ICE-DT-entropy to ICE-CS. Each point in the diagrams correspond to a program in Table 1. Points above the diagonal indicate that ICE-DT-* is faster than ICE-CS.

- The most important observation is that *the decision tree learners converge and successfully find an adequate invariant for all programs.* We were surprised at this convergence; we expected heuristic machine learning to not necessarily converge; however, the outer ICE learning guidance from the teacher seems to make these learning algorithms fairly robust.
- The constraint solver-based learner ICE-CS times out on five programs. There are several programs where ICE-CS converges, but takes much longer than

[2] Available at `http://web.engr.illinois.edu/~garg11/dtree/benchmarks.zip`

Program	White-box	Black-box								
	CPA-checker	ICE-CS			ICE-DT-entropy			ICE-DT-penalty		
		P,N,I	#R	T(s)	P,N,I	#R	T(s)	P,N,I	#R	T(s)
afnp	**Timeout**	1,19,15	29	3.6	1,3,10	14	0.7	1,2,8	11	0.5
array	×	4,7,11	14	0.5	15,16,67	95	6.3	4,6,16	25	1.4
arrayn	×	4,7,5	7	0.3	2,1,2	5	0.2	2,1,2	5	0.2
arrayn.v	×	6,8,6	8	0.3	3,3,1	6	0.3	3,3,1	6	0.7
cegar1	0.1	1,1,1	3	0.0	3,1,1	5	0.2	3,1,1	5	0.2
cegar1.v	0.1	1,1,1	3	0.0	7,2,1	9	0.4	7,2,1	9	0.4
cegar2	**0.5**	**4,20,14**	**28**	**9.5**	**5,10,8**	**23**	**1.0**	**5,9,9**	**22**	**1.0**
cegar2.v	**0.6**	**Timeout**			**11,50,33**	**89**	**5.9**	**12,36,12**	**55**	**2.0**
cggmp05	**0.4**	**1,36,50**	**71**	**51.1**	**1,12,55**	**68**	**4.6**	**1,17,50**	**68**	**7.2**
dec	**Timeout**	1,1,1	3	0.0	1,2,0	3	0.4	1,2,0	3	0.2
decn.v	×	2,4,3	7	0.1	5,4,1	9	0.3	5,4,1	9	0.3
ex14	0.2	2,5,1	7	0.0	1,1,0	2	0.1	1,1,0	2	0.1
ex14n	0.1	2,2,1	4	0.0	1,1,0	2	0.0	1,1,0	2	0.1
ex14n.v	0.1	8,8,7	10	0.1	3,4,0	5	0.5	3,4,0	5	0.2
ex23	**Timeout**	**5,32,40**	**69**	**17.5**	**5,21,12**	**34**	**2.7**	**5,8,1**	**11**	**0.6**
ex23.v	**Timeout**	**14,39,53**	**78**	**48.7**	**9,14,4**	**20**	**1.0**	**11,16,49**	**68**	**4.1**
ex7	0.1	1,2,1	2	0.0	1,1,0	2	0.1	1,1,0	2	0.1
ex7.v	0.2	1,2,1	2	0.0	1,1,0	2	0.1	1,1,0	2	0.1
fig1	4.4	2,5,1	6	0.1	2,4,1	6	0.2	2,4,1	6	0.4
fig1.v	6.4	6,11,5	14	0.5	4,5,1	8	0.4	4,6,1	9	0.3
fig3	0.1	2,4,2	6	0.1	4,5,0	6	0.3	4,5,0	6	0.2
fig3.v	0.2	5,8,5	8	0.2	5,7,0	8	0.4	5,7,0	8	0.4
fig9	0.1	0,2,0	2	0.0	1,1,0	2	0.1	1,1,0	2	0.1
fig9.v	0.1	0,2,0	2	0.0	1,1,0	2	0.0	1,1,0	2	0.1
form22	0.6	1,18,11	22	1.8	1,13,34	48	2.6	1,8,32	41	1.9
form25	**0.7**	**1,46,30**	**49**	**14.0**	**1,71,6**	**78**	**3.7**	**1,62,4**	**67**	**3.0**
form27	**2.0**	**Timeout**			**1,142,13**	**156**	**9.5**	**1,104,15**	**120**	**5.8**
inc	**Timeout**	**3,12,101**	**112**	**1.7**	**9,6,104**	**117**	**6.5**	**5,3,100**	**106**	**4.9**
incn	×	3,4,3	8	0.1	3,3,1	7	0.3	4,3,3	10	0.4
incn.v	×	9,10,6	11	0.2	4,3,1	7	0.2	5,4,1	8	0.3
matrix1	×	2,9,3	8	0.3	5,5,2	8	0.5	5,5,2	8	0.5
matrix1n	×	4,12,4	8	0.9	4,9,2	8	0.5	5,11,2	9	0.7
matrix1n.v	×	2,13,4	7	0.9	6,13,1	10	0.7	8,19,2	14	1.0
matrix2	×	**8,19,13**	**27**	**22.9**	**9,10,7**	**24**	**1.3**	**9,10,5**	**22**	**1.2**
matrix2n	×	**Timeout**			**18,52,38**	**107**	**9.8**	**17,36,16**	**66**	**3.5**
matrix2n.v	×	**Timeout**			**32,60,11**	**101**	**8.4**	**17,73,19**	**101**	**8.3**
nc11	0.8	5,15,7	18	0.7	3,6,5	13	0.6	2,4,4	9	0.4
nc11n	×	4,6,3	10	0.4	3,3,3	8	0.4	3,3,3	8	0.3
nc11n.v	×	7,11,7	13	0.9	10,13,3	17	0.7	8,9,3	13	0.7
sum1	×	2,15,10	17	2.3	4,17,3	21	3.3	3,13,3	17	1.1
sum1.v	×	**Timeout**			**5,93,75**	**169**	**18.7**	**6,20,5**	**26**	**3.2**
sum3	0.0	1,3,1	4	0.1	1,4,1	6	0.3	1,4,1	6	0.3
sum3.v	0.1	5,8,6	8	0.2	5,8,0	9	0.4	5,8,0	9	0.4
sum4	1.6	1,23,31	44	3.5	1,9,44	54	3.1	1,7,38	46	2.5
sum4n	×	**6,29,21**	**34**	**11.6**	**5,14,5**	**20**	**1.1**	**5,13,3**	**17**	**0.9**
sum4n.v	×	**12,33,24**	**38**	**31.8**	**6,27,12**	**40**	**2.3**	**8,40,10**	**51**	**2.8**
tacas	0.1	7,8,5	14	1.7	10,7,4	19	0.9	10,7,4	20	0.9
tacas.v	0.1	10,11,9	17	2.4	18,14,5	28	1.4	16,16,5	29	3.8
trex01	0.0	2,3,0	3	0.0	2,3,0	5	0.2	2,3,0	5	0.2
trex01.v	0.1	2,3,0	3	0.0	3,3,0	4	0.3	3,3,0	4	0.2
trex3	×	6,19,6	19	2.7	7,12,4	20	1.2	4,8,3	12	0.7
trex3.v	×	**12,25,12**	**25**	**10.8**	**12,8,2**	**19**	**3.5**	**10,8,2**	**16**	**0.9**
vsend	0.0	1,1,0	2	0.0	1,1,0	2	0.1	1,1,0	2	0.1
w1	×	1,3,3	5	0.0	2,1,1	4	0.4	2,2,1	5	0.2
w2	×	2,4,1	4	0.0	2,2,1	5	0.3	1,2,1	4	0.2
Aggregate	**Timeouts = 5**	**Timeouts = 5 Time = 244s**			**Timeouts = 0 Time = 57s**			**Timeouts = 0 Time = 49s**		

Table 1. Results comparing different invariant synthesis tools. A × indicates that the tool incorrectly reported an error (false negative); P, N, I are the number of positive, negative examples and implications in the final sample of the learners; $\#R$ is the number of rounds and T is the time in seconds (Timeout was $60\,s$).

the decision-tree learners (e.g., for programs *cggmp05, ex23.v, matrix2*). It is easy to observe this comparison in Figure 2 where we plot the run time of the decision tree learners and compare them to the time taken by ICE-CS.

In Table 1 we do not give results for the decision tree learner that completely ignores implications while choosing the attributes to split the sample. Ignoring implications is not a good idea and there are simple natural examples where this learner does not converge. We also observed that this learner led to 20% larger conjectures (measured as the total number of nodes in the conjectured decision trees) as compared to the other two decision tree learners, averaged over all rounds for all programs. To check the robustness of the various black-box learners with the number of variables in the program, we generated program variations by adding *three* extra variables that are havoc-ed inside the loop. These programs have the suffix ".v". Our decision tree learners usually perform equally well for programs with these extra variables whereas ICE-CS times outs on several of these programs (*cegar2, sum1, matrix2n*). Adding variables increases the search space for the black-box learners and our decision tree algorithms turn out to be better with regards to generalization and sifting and picking the relevant variables. Finally, note that since our teacher returns implications only if it cannot find positive/negative examples, many programs, where the final sample includes implications, needed implications to find the invariant.

Comparison With Randomized Search: Recently, Sharma et. al [44] have built an ICE learner based on randomized search for synthesizing program invariants. As the tool is based on random search, the time to learn an invariant using [44] can span a large range (for e.g., for programs such as *cegar2* and *fig1*, time to learn an adequate invariant varied from a second to more than ten minutes). Additionally, the tool searches for numerical invariants that have constant thresholds that belong to a small bag of values (typically of size three to five) that are mined from the program code. Our learner in comparison searches a much larger space since it learns arbitrary thresholds, and finds invariants even when the constants needed are not present in the program, as in the programs *cggmp05* and *ex23*.

6 Conclusion

We have presented a promising machine learning technique of learning decision trees from positive, negative, and implication counter-examples that can be used to efficiently synthesize invariants expressed as Boolean combinations of predicates over numerical and Boolean attributes. We believe that our work opens up the gateway for adapting more machine learning algorithms for various other learning domains for invariant generation, by adapting them to ICE. The impressive performance of our learner in weeding out the irrelevant attributes and quickly choosing the ones that matter suggest that machine learning techniques may even be useful in the simpler Houdini-based invariant synthesis of purely *conjunctive* formulae, in applications such as verification of GPU programs [8]. Our decision tree learning also works for arbitrary predicates, and hence we would like to use them to synthesize other kinds of invariants, such as data-structure invariants using separation logic.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 1–17. IEEE (2013), http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679385
2. Alur, R., Cerný, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005. pp. 98–109. ACM (2005), http://doi.acm.org/10.1145/1040305.1040314
3. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: Etessami, K., Rajamani, S.K. (eds.) Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3576, pp. 548–562. Springer (2005), http://dx.doi.org/10.1007/11513988_52
4. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987), http://dx.doi.org/10.1016/0890-5401(87)90052-6
5. BakIr, G., Hofmann, T., Scholkopf, B., Smola, A.J., Taskar, B., Vishwanathan, S.: Predicting Structured Data. MIT Press, Cambridge, MA, USA (2007)
6. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Burke, M., Soffa, M.L. (eds.) Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001. pp. 203–213. ACM (2001), http://doi.acm.org/10.1145/378795.378846
7. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005), http://dx.doi.org/10.1007/11804192_17
8. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: Gpuverify: A verifier for gpu kernels. SIGPLAN Not. 47(10), 113–132 (Oct 2012), http://doi.acm.org/10.1145/2398857.2384625
9. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011), http://dx.doi.org/10.1007/978-3-642-22110-1_16
10. Biere, A., Bloem, R. (eds.): 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, Lecture Notes in Computer Science, vol. 8559. Springer (2014), http://dx.doi.org/10.1007/978-3-319-08867-9
11. Bradley, A.R.: Sat-based model checking without unrolling. In: Jhala, R., Schmidt, D.A. (eds.) Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6538, pp. 70–87. Springer (2011), http://dx.doi.org/10.1007/978-3-642-18275-4_7

12. Brat, G., Rungta, N., Venet, A. (eds.): NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings, Lecture Notes in Computer Science, vol. 7871. Springer (2013), http://dx.doi.org/10.1007/978-3-642-38088-4
13. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2619, pp. 331–346. Springer (2003), http://dx.doi.org/10.1007/3-540-36577-X_24
14. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: Jr. and Somenzi [29], pp. 420–432, http://dx.doi.org/10.1007/978-3-540-45069-6_39
15. Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning 20(3), 273–297 (1995), http://dx.doi.org/10.1007/BF00994018
16. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 238–252. ACM (1977), http://doi.acm.org/10.1145/512950.512973
17. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978. pp. 84–96. ACM Press (1978), http://doi.acm.org/10.1145/512760.512770
18. Ernst, M.D., Czeisler, A., Griswold, W.G., Notkin, D.: Quickly detecting relevant program invariants. In: Ghezzi, C., Jazayeri, M., Wolf, A.L. (eds.) Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000. pp. 449–458. ACM (2000), http://doi.acm.org/10.1145/337180.337240
19. Filé, G., Ranzato, F.: The powerset operator on abstract interpretations. Theor. Comput. Sci. 222(1-2), 77–111 (1999), http://dx.doi.org/10.1016/S0304-3975(98)00007-3
20. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for esc/java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2021, pp. 500–517. Springer (2001), http://dx.doi.org/10.1007/3-540-45251-6_29
21. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Abstract interpretation over non-lattice abstract domains. In: Logozzo and Fähndrich [32], pp. 6–24, http://dx.doi.org/10.1007/978-3-642-38856-9_3
22. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 813–829. Springer (2013), http://dx.doi.org/10.1007/978-3-642-39799-8_57

23. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: Biere and Bloem [10], pp. 69–87, http://dx.doi.org/10.1007/978-3-319-08867-9_5
24. Garoche, P., Kahsai, T., Tinelli, C.: Incremental invariant generation using logic-based automatic abstract transformers. In: Brat et al. [12], pp. 139–154, http://dx.doi.org/10.1007/978-3-642-38088-4_10
25. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008. pp. 281–292. ACM (2008), http://doi.acm.org/10.1145/1375581.1375616
26. Gupta, A., Rybalchenko, A.: Invgen: An efficient invariant generator. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 634–640. Springer (2009), http://dx.doi.org/10.1007/978-3-642-02658-4_48
27. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Launchbury, J., Mitchell, J.C. (eds.) Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002. pp. 58–70. ACM (2002), http://doi.acm.org/10.1145/503272.503279
28. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Hermanns, H., Palsberg, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings. Lecture Notes in Computer Science, vol. 3920, pp. 459–473. Springer (2006), http://dx.doi.org/10.1007/11691372_33
29. Jr., W.A.H., Somenzi, F. (eds.): Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings, Lecture Notes in Computer Science, vol. 2725. Springer (2003)
30. Kawaguchi, M., Rondon, P.M., Jhala, R.: Type-based data structure verification. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009. pp. 304–315. ACM (2009), http://doi.acm.org/10.1145/1542476.1542510
31. Littlestone, N.: Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. Machine Learning 2(4), 285–318 (1987), http://dx.doi.org/10.1007/BF00116827
32. Logozzo, F., Fähndrich, M. (eds.): Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings, Lecture Notes in Computer Science, vol. 7935. Springer (2013), http://dx.doi.org/10.1007/978-3-642-38856-9
33. McMillan, K.L.: Interpolation and sat-based model checking. In: Jr. and Somenzi [29], pp. 1–13, http://dx.doi.org/10.1007/978-3-540-45069-6_1
34. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4144, pp. 123–136. Springer (2006), http://dx.doi.org/10.1007/11817963_14
35. Miné, A.: The octagon abstract domain. In: WCRE. p. 310 (2001), http://computer.org/proceedings/wcre/1303/13030310abs.htm

36. Mitchell, T.M.: Machine learning. McGraw Hill series in computer science, McGraw-Hill (1997)
37. Neider, D.: Applications of Automata Learning in Versification and Synthesis. Ph.D. thesis, RWTH Aachen University (April 2014)
38. Neider, D., Jansen, N.: Regular model checking using solver technologies and automata learning. In: Brat et al. [12], pp. 16–31, http://dx.doi.org/10.1007/978-3-642-38088-4_2
39. Quinlan, J.R.: Induction of decision trees. Machine Learning 1(1), 81–106 (1986), http://dx.doi.org/10.1023/A:1022643204877
40. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
41. Rosenblatt, F.: The perceptron: a probabilistic model for information storage and organization in the brain. Psychological Review 65(6), 386–408 (1958)
42. Sankaranarayanan, S., Ivancic, F., Shlyakhter, I., Gupta, A.: Static analysis in disjunctive numerical domains. In: Yi, K. (ed.) Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4134, pp. 3–17. Springer (2006), http://dx.doi.org/10.1007/11823230_2
43. Shannon, C.E.: A mathematical theory of communication. The Bell System Technical Journal 27, 379–423, 623–656 (1948), http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf
44. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: Biere and Bloem [10], pp. 88–105, http://dx.doi.org/10.1007/978-3-319-08867-9_6
45. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 574–592. Springer (2013), http://dx.doi.org/10.1007/978-3-642-37036-6_31
46. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Nori, A.V.: Verification as learning geometric concepts. In: Logozzo and Fähndrich [32], pp. 388–411, http://dx.doi.org/10.1007/978-3-642-38856-9_21
47. Sharma, R., Nori, A.V., Aiken, A.: Interpolants as classifiers. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. Lecture Notes in Computer Science, vol. 7358, pp. 71–87. Springer (2012), http://dx.doi.org/10.1007/978-3-642-31424-7_11
48. Sharma, R., Nori, A.V., Aiken, A.: Bias-variance tradeoffs in program analysis. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 127–138. ACM (2014), http://doi.acm.org/10.1145/2535838.2535853
49. Solar Lezama, A.: Program Synthesis By Sketching. Ph.D. thesis, EECS Department, University of California, Berkeley (Dec 2008), http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html
50. Thakur, A., Lal, A., Lim, J., Reps, T.: Posthat and all that: Attaining most-precise inductive invariants. Tech. Rep. TR1790, University of Wisconsin, Madison, WI (Apr 2013)

51. Vardhan, A., Viswanathan, M.: Learning to verify branching time properties. Formal Methods in System Design 31(1), 35–61 (2007), http://dx.doi.org/10.1007/s10703-006-0026-x

A Illustrating Why Good Splits Might Turn Into Bad Ones if Implications are Ignored

Example 1. Suppose that Algorithm 2 processes the sample shown in Figure 3a, which also depicts the (only) implication in the global set *Impl*.

(a) Sample before splitting (b) Samples after splitting

Fig. 3. The samples discussed in Example 1.

When using the original split procedure (i.e., using information gain while ignoring the implication and its corresponding unclassified data points), the learner splits the sample with respect to attribute x at threshold $c = 3$ since this split yields the highest information gain—the information gain is 1 since the entropy of the resulting two subsamples is 0. Using this split, the learner partitions the sample into $\mathit{Examples}_a$ and $\mathit{Examples}_{\neg a}$ and recurses on $\mathit{Examples}_a$. Since $\mathit{Examples}_a$ contains only unclassified and positively classified points, it turns all unclassified points in this sample positive and propagates this information along the implication. This results in the situation depicted in Figure 3b. Note that the learner now needs to split $\mathit{Examples}_{\neg a}$ since the unclassified data points in it are now classified positively.

On the other hand, considering the implication and its corresponding data points allows splitting the sample such that the antecedent and the consequent of the implication both belong to either $\mathit{Examples}_a$ or $\mathit{Examples}_{\neg a}$ (e.g., with respect to x and threshold $c = 2$). Such a split has the advantage that no further splits are required and, hence, results in a smaller tree. ⌟

B Proof of Lemma 1

We now argue why Algorithm 2 always results in a terminating procedure that constructs a decision tree consistent with the sample. For this we first introduce a property of the original sample S and the partial valuation for the implication end-points G, called a *valid sample*. In the description below, we refer to $S \cup G$ as the combined valuation defined on the points by S^+, S^-, and G.

Definition 1 (Valid sample). *A sample S is* valid *if for every implication $(x, y) \in S^{\Rightarrow}$*

– *it is not the case that x is classified positively and y negatively in $S \cup G$;*
– *it is not the case that x is classified positively and y is unclassified in $S \cup G$;*
– *it is not the case that y is classified negatively and x is unclassified in $S \cup G$.*

A valid sample has the following interesting property.

Lemma 1. For any valid sample (with partially classified end-points of implications), extending it by classifying all unclassified points as positive will result in a consistent classification, and extending it by classifying all unclassified points as negative will also result in a consistent classification.

Proof. The above lemma is easy to prove: consider the extension of a valid sample by classifying all unclassified points as positive. Assume, by way of contradiction, that this valuation inconsistent. Then, there is some implication pair (x, y) such that x is classified as positive and y is classified as negative. Since such an implication pair could not have already existed in the valid sample (by definition), it must have been caused by the extension. Since we introduced only positive classifications, it must have been that x (and not y) is the only new classification. Hence the valid sample must have had the implication pair (x, y) with y classified as negative and x being unclassified, which contradicts the definition of a valid sample. The proof of the extension using negative classifications can be shown using a similar argument.

Note that Algorithm 2 always maintains a valid sample. When we have a valid sample and are at a leaf where the algorithm is working on a subsample that has only positive points and unclassified points, it would classify all these unclassified points to be positive. Since the extension only causes positive classifications, the sample cannot be inconsistent (classifying all points as positive, after all, will satisfy all implications, by Lemma 1). Moreover, at this point we will take the implication closure of the new positively classified points and hence maintain a valid sample. A similar argument holds when the algorithm creates a negative leaf node.

The above argument shows that our algorithm will never end up with an inconsistent sample, which prove the correctness of Algorithm 2. Moreover, all attributes are assumed to be numerical and can always split any set with more than one element, which proves termination as well.