D^3: Data-Driven Disjunctive Abstraction

Hila Peleg[1], Sharon Shoham[2], and Eran Yahav[1]

[1] Technion, {hilap,yahave}@cs.technion.ac.il
[2] The Academic College of Tel Aviv-Yaffo, sharon.shoham@gmail.com

Abstract. We address the problem of computing an abstraction for a set of examples, which is precise enough to separate them from a set of counterexamples. The challenge is to find an over-approximation of the positive examples that does not represent any negative example. Conjunctive abstractions (e.g., convex numerical domains) and limited disjunctive abstractions, are often insufficient, as even the best such abstraction might include negative examples. One way to improve precision is to consider a general disjunctive abstraction.
We present D^3, a new algorithm for learning general disjunctive abstractions. Our algorithm is inspired by widely used machine-learning algorithms for obtaining a classifier from positive and negative examples. In contrast to these algorithms which cannot generalize from disjunctions, D^3 obtains a disjunctive abstraction that minimizes the number of disjunctions. The result generalizes the positive examples as much as possible without representing any of the negative examples. We demonstrate the value of our algorithm by applying it to the problem of data-driven differential analysis, computing the abstract semantic difference between two programs. Our evaluation shows that D^3 can be used to effectively learn precise differences between programs even when the difference requires a disjunctive representation.

1 Introduction

We address the problem of computing an abstraction for a set of examples, which is precise enough to separate them from a set of counterexamples. Given a set of positive examples C^+ and a set of negative examples C^-, both drawn from some concrete domain $\mathcal{D}$, our goal is to compute an abstraction of C^+ using a *disjunctive* abstract domain, such that the abstraction overapproximates C^+, but does not represent any example from C^-.

The need for such an abstraction arises in many settings [32, 5, 13], including the problem of differential analysis - computing the abstract semantic difference between two programs [28, 29, 35]. The abstract semantic difference between two programs often contains ranges of input values for which the programs are known to produce the same outputs, but other ranges for which the output values differ. Computing a safe abstraction of difference/similarity ranges can produce a succinct description of the difference/similarity between programs.

Unfortunately, computing such an abstraction is tricky due to the delicate interplay between generalization and precision (required to ensure that the abstraction is safe). When there are multiple ranges of equivalence or difference, typical conjunctive abstractions (e.g., convex numerical domains [24, 11]) and limited disjunctive abstractions [23, 30, 6, 3, 15], are often insufficient, as even the best such abstraction might include negative examples. On the other hand, general (unlimited) disjunctive abstractions are too precise and do not naturally generalize.

We present D^3, a new Data-Driven algorithm for learning general Disjunctive abstractions. D^3 is an active learning algorithm that iteratively accepts an example and its label as positive or negative, and incrementally updates the disjunctive abstraction of all examples seen. D^3 is driven by a new notion of *safe generalization* used to compute the abstraction of the seen examples. Safe generalization *generalizes* a precise disjunctive abstraction of the positive examples into a more abstract one, but does so in a *safe* way that does not represent any negative example.

The exploration of the input space is directed by D^3 by restricting the sampling to advantageous regions of the space derived from the intermediate abstractions.

D^3 is a general algorithm and can be instantiated with different choices for the following: (i) an *oracle* responsible for picking the next sample input from a given region, (ii) an implementation of a *teacher*, used to label each sample, and (iii) the abstract domain over which disjunctive abstractions are computed.

To implement differential analysis, we instantiate D^3 with a code-aware oracle for picking the next input, a teacher that labels an input by executing both programs and comparing outputs, and several abstractions including intervals, congruence intervals, and boolean predicates over arrays.

The main contributions of this paper are:

- A new operation, *safe generalization*, which takes a disjunctive abstraction and generalizes it further while avoiding describing a set of counterexamples.
- A new algorithm D^3 for learning general disjunctive abstractions, which uses safe generalization, as well as a strategy to direct exploration of the input space.
- An implementation of D^3 and its application to the problem of data-driven differential analysis, computing the abstract semantic difference between two programs. Our evaluation shows that D^3 can be used to effectively learn precise differences between programs even when the difference requires a disjunctive representation.

2 Overview

In this section, we provide an informal overview of our approach using a differential analysis example. Fig. 1 shows two functions computing the sum of digits in a number.

Fig. 1(a) is a model Scala implementation for summing the digits of an input number. Fig. 1(b) is an implementation by a less experienced programmer that uses a loop construct rather than the tail recursive approach. While the second implementation is very similar to a correct implementation, it suffers from an incorrect initialization of the result variable, which is easily missed with poor testing.

The goal of differential analysis is to compute an abstract representation of the difference between programs. For the programs of Fig. 1, the difference can be described as $\bigvee_{i \in \{1..9\}} (x \mod 10 = i) \wedge (x \leq -11 \vee x \geq 11)$. The similarity between these two programs (inputs for which the programs agree) can be described as $(x \mod 10 = 0) \vee (-9 \leq x \leq 9)$.

We use an *active learning* approach for computing the difference between the programs. In active learning, a *learner* iteratively picks points and asks a *teacher* for the

```
def sumOfDigits(x : Int) : Int = {
  @tailrec def sodRec(
                sum : Int,
                rest : Int) : Int = {
    if (rest == 0) sum
    else sodRec(sum + rest % 10, rest/10)
  }
  sodRec(0,Math.abs(x))
}
```

(a)

```
def sumOfDigitsWrong(x : Int) : Int = {
  var y = Math.abs(x)
  if (y < 10) y
  else {
    var sum = y % 10
    while (y > 0) {
          sum += y % 10
          y = y / 10
    }
    sum
  }
}
```

(b)

Fig. 1: Two Scala functions for computing the sum of a number's digits. (a) is a correct implementation. (b) has an error in initializing the variable `sum` and is correct only on numbers that have 0 as the least significant digit, or on single-digit numbers.

classification of each point. The result of active learning is a *classifier* that generalizes from the observed points and can be used to classify new points.

In our example, the learner is trying to learn the difference between two programs P and P'. We provide a simple teacher that runs the programs and classifies a given input point c as "positive" when both programs produce the same result, i.e. $P(c) = P'(c)$, and "negative" when the results of the two programs differ, i.e. $P(c) \neq P'(c)$.

Our starting point is the *Candidate Elimination* algorithm, presented formally in the next section. Candidate Elimination proceeds iteratively as follows: in each iteration of the algorithm, the learner picks a point to be classified, asks the teacher for a classification, and updates an internal representation that captures the classification that has been learned so far. Based on this internal representation, the learner can pick the next point to be classified. The iterative process is repeated until the generalization of the positive points and the exclusion of the negative points yields the same representation.

Applying the algorithm to our example program yields the following points:

$$(0, pos), (7, pos), (10, pos), (60, pos), (47, neg), (73, neg), (88, neg)$$

The challenge is how to internally represent the set of positive points and the set of negative points. The set of positive points cannot be directly represented using a conjunctive (convex) representation, as the range $[0, 60]$ also includes the negative point 47. On the other hand, the negative range $[47, 88]$ also includes the positive point 60.

Trying to represent the positive points using a precise disjunctive representation yields no generalization in the algorithm (Section 3.2), and would yield the formula: $x = 0 \vee x = 7 \vee x = 10 \vee x = 60$. This disjunction would grow as additional positive points are added, does not provide any generalization for points that have not been seen, and cannot represent an unbounded number of points.

The D^3 Algorithm The main idea of the D^3 algorithm (Algorithm 2) is to incrementally construct a generalized disjunctive representation for the positive and negative examples. Technically, D^3 operates by maintaining two formulas: φ_{pos} that maintains the generalized disjunction representing positive examples, and φ_{neg} that maintains the generalized disjunction representing the negative examples. The algorithm preserves the invariant that φ_{pos} and φ_{neg} both correctly classify all seen points. That is, any seen positive point satisfies φ_{pos}, and any seen negative point satisfies φ_{neg}. When a new point arrives, D^3 uses the generalization of the conjunctive domain as much as possible, but uses disjunctions when needed in order to exclude points of opposite classification.

In the differential analysis setting, φ_{pos} attempts to describe the similarity between programs and φ_{neg} attempts to describe the difference. For the example points above, the algorithm constructs the following φ_{pos}: $(7 \leq x \leq 7 \wedge x \mod 10 = 7) \vee (0 \leq x \leq 60 \wedge x \mod 10 = 0)$. Note that this representation correctly generalizes to include the positive points $20, 30, 40, 50$ that were not seen. The resulting φ_{neg} is $(47 \leq x \leq 47 \wedge x \mod 10 = 7) \vee (73 \leq x \leq 88)$.

The existence of points that satisfy both φ_{neg} and φ_{pos} does not contradict the invariant of the algorithm because both formulas include unseen points due to generalization. In fact, the points in the intersection can be used to *refine* the generalization. Technically, this is done by using the intersection as one of the regions to be sampled.

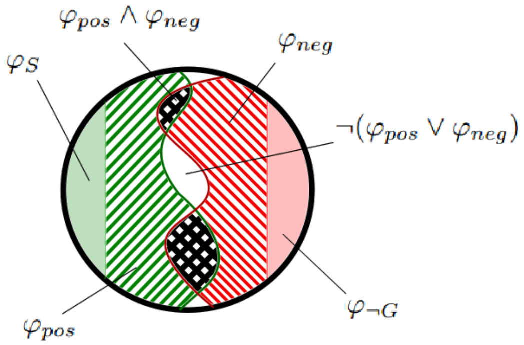
$\varphi_{pos} \wedge \varphi_{neg}$
φ_{neg}
φ_S
$\neg(\varphi_{pos} \vee \varphi_{neg})$
φ_{pos}
$\varphi_{\neg G}$

Fig. 2: The regions of the input space as seen by the D^3 algorithm

In addition to φ_{pos} and φ_{neg}, the algorithm maintains φ_S and $\varphi_{\neg G}$, the precise disjunctive representations of the positive and negative examples, respectively. Together, the four formulas determine the regions to be sampled, as depicted in Fig. 2:

- Uncovered: $\neg(\varphi_{pos} \vee \varphi_{neg})$
- Covered disagreement: $\varphi_{pos} \wedge \varphi_{neg}$
- Positive abstracted disagreement: $\varphi_{pos} \wedge \neg\varphi_S$
- Negative abstracted disagreement: $\varphi_{neg} \wedge \neg\varphi_{\neg G}$

The *covered* and *uncovered* are regions where a given point would either satisfy both φ_{pos} and φ_{neg}, or neither. The *positive abstract disagreement* region is where a point would satisfy the generalized disjunctive representation φ_{pos} but not the precise disjunctive representation φ_S (that is, the point is the result of generalization). The *negative abstract disagreement* plays a similar role for φ_{neg} and $\varphi_{\neg G}$.

Sampling from each of these regions ensures the algorithm would progress towards abstracting and refining both positive and negative generalizations. Convergence will occur if φ_{pos} and $\neg\varphi_{neg}$ are equivalent, which means covered disagreement is eliminated, and no region of the space is uncovered.

3 Active Concept Learning

Concept learning is an area of machine learning dedicated to learning a classifier that is an abstraction of a dataset using a predefined language of predicates. This section details the most commonly used concept learning algorithm, Candidate Elimination, and its relation to abstract domains. We further discuss the limitations of Candidate Elimination, which are later addressed by our new algorithm.

Concept learning Concept learning aims at learning a concept in a given *concept language*. In our setting, a concept language would be used to describe the space of inputs to a program. From now on, we fix an input space, denoted $\mathcal{D}$ (also called a *domain*).

Definition 1 (concept language). *A* concept *of domain $\mathcal{D}$ is a boolean function a over $\mathcal{D}$. i.e. $a : \mathcal{D} \to \{true, false\}$. An element $c \in \mathcal{D}$ is* described by *the concept a if $a(c) = true$. A* concept language *L is a set of concepts, i.e. $L \subseteq \{true, false\}^{\mathcal{D}}$.*

Each concept describes a subset of $\mathcal{D}$, and a concept language defines the set of possible subsets available to describe the domain. A concept language is usually defined by a set of possible *descriptions* (templates) of boolean functions.

Example 1. The concept language of intervals includes all concepts described as $[l, h] = \lambda x.l \leq x \leq h$ s.t. $l, h \in \mathbb{N}$. $[0, 42]$ is a concept in the intervals concept language, which from a domain of integers describes the subset $\{0, 1, \dots, 42\}$.

Concept languages based on logical formulas Given a concept language L_0, we view its concepts (which are boolean functions) as *atoms* over which propositional formulas can be constructed using logical connectives, such as negation, conjunction and disjunction, thus defining new concepts (boolean functions). For example, if $a_1, a_2 \in L_0$, then the formula $\varphi = a_1 \wedge a_2$ represents the function $\lambda x.\ a_1(x) \wedge a_2(x)$. Note that this boolean function need not be in the original concept language L_0. Thus, we obtain new, richer, concept languages.

Definition 2 (Conjunctive concepts). *Given a concept language L_0,* conjunctive concepts over *L_0 (or simply* conjunctive concepts*) are concepts defined by a conjunction of finitely many concepts from L_0.*

A cartesian product $L_1 \times \ldots \times L_n$ is a special case of a conjunctive concept language over $L_0 = \bigcup_{1 \leq i \leq n} L_i$, where the concepts are tuples comprised of one concept from each L_i, with the meaning of conjunction.

For example, the concept language of rectangles in 2D over a domain consisting of pairs (x, y), is the product of two interval concept languages, one bounding the x axis and the other bounding the y axis, and therefore it is a conjunctive concept language.

Disjunctive concepts are defined similarly to conjunctive concepts. A disjunctive concept language over L_0 corresponds to the powerset domain over L_0 [10]. We therefore denote it $\mathcal{P}(L_0)$.

Concept lattices Concept learning algorithms such as Candidate Elimination [25] are based on the fact that every concept language L has an inherent partial order, denoted $\preceq$, based on the implication relation between the individual concepts, defined in [25] as the *more specific than* relation. Formally, $a_1 \preceq a_2$ if and only if for every $c \in \mathcal{D}$, $a_1(c) \Rightarrow a_2(c)$. For example, $c \in [1,4] \Rightarrow c \in [0,80]$ which means $[1,4] \preceq [0,80]$.

We are particularly interested in cases where this partially ordered set, $(L, \preceq)$, forms a lattice. We assume that all concept languages include $\bot = \lambda x.false$ and $\top = \lambda x.true$, which are the least and greatest concepts w.r.t. $\preceq$, respectively. For instance, in the intervals lattice, $[1,3] \sqcup [5,8] = [1,8]$, and $[1,3] \sqcap [5,8] = \bot$.

Concepts as abstractions We view a concept language L as an *abstract domain* for $\mathcal{D}$, accompanied by a concretization function $\gamma : L \to 2^{\mathcal{D}}$ that transforms a concept $a \in L$ into all of its described objects, and an abstraction function $\beta : \mathcal{D} \to L$ which transforms an element $c \in \mathcal{D}$ to the *most specific concept representation*.[3] In the intervals concept language, for example, $\beta(c) = [c,c]$ for every $c \in \mathcal{D}$. Note that by definition of the $\preceq$ relation, $a_1 \preceq a_2 \iff \gamma(a_1) \subseteq \gamma(a_2)$.

3.1 Candidate Elimination

Candidate Elimination is a machine learning algorithm aimed at learning a binary classifier from $\mathcal{D}$ to the categories "positive" and "negative". The input to the algorithm is a set of positive examples $C^+ \subseteq \mathcal{D}$ and a set of negative examples $C^- \subseteq \mathcal{D}$. The output is a classifier, given as a concept, also called *hypothesis*, that is consistent with all the examples.

Definition 3 (consistency). *A hypothesis h is* consistent *with a set of positive examples C^+ and a set of negative examples C^- if and only if for every $c \in C^+ \cup C^-$, $h(c) = true \iff c \in C^+$.*

The Candidate Elimination algorithm holds a lower bound and an upper bound of possible consistent hypotheses in the lattice, representing all the lattice elements inbetween. Every concept below the upper bound excludes all the concrete points the upper bound excludes, and every concept above the lower bound includes all points that the lower bound includes. The hypotheses represented by the upper and lower bound created by processing a concrete set $C = C^+ \cup C^-$ are called the *version space* of C.

Algorithm 1 describes the full Candidate Elimination algorithm. In the code, we use a function $label(x)$ which for $x \in \mathcal{D}$ returns either "positive" or "negative". In the case of predefined sets of points C^+, C^-, $label$ is a partial function defined for $C^+ \cup C^-$ that will return positive if and only if $x \in C^+$. In the active learning case, it will compute the label for any point in $\mathcal{D}$. In this case it will also be called a *teacher*.

The algorithm starts with a specific (lower) bound, $S = \bot$[4] and a set of generic (upper) bounds $G = \{\top\}$ (every element in G is a possible generic bound). Using concrete examples from the sets C^+ and C^-, the algorithm advances its hypotheses

[3] A most specific representation need not exist. For simplicity of the presentation, we consider the case where it does, and explain what adaptations are needed when it does not.

[4] If β maps a concrete point to a single concept which best represents it, it is easily shown that it suffices to maintain S as a single element. Candidate Elimination can also handle multiple representations, in which case S will be a *set* of specific bounds, similarly to G.

Algorithm 1: The Candidate Elimination algorithm formulated in abstract domain operations

```
$S \leftarrow \bot$
$G \leftarrow \{\top\}$
for $c \leftarrow Samples$ do
    if $label(c)$ is positive then $S \leftarrow S \sqcup \beta(c)$
    else $G \leftarrow \{g \sqcap n \mid g \in G, n \in comp^{-}(\{c\})\}$
    $G \leftarrow \{g \in G \mid S \sqsubseteq g\}$
    if $G = \emptyset$ then
        return $\bot$
if $S \in G$ then return $S$
// Training examples ran out but $S$ and $G$ have not converged
return some hypothesis bound between $S$ and $G$
```

bounds from either direction until the lower and upper bound converge. For any positive example c, the algorithm modifies S to include c, and for every negative example c', it modifies *all* the bounds in G to eliminate concepts that include c'. If the concept language used is a lattice, it is easy to describe the Candidate Elimination algorithm in terms of lattice operations. Modifying the bounds to include and exclude examples is done with the join and meet operations, walking through the implication lattice.

The increase of the specific bound uses the abstraction function β. In order to describe the lowering of the generic bound, we define the set which is the underapproximated complementation of a set, $comp^{-}$.

Definition 4 (underapproximated complementation). *Given a set of concrete points $C \subseteq \mathcal{D}$, $comp^{-}(C)$ is the underapproximating complement of C. $comp^{-}(C) \subseteq L$ s.t.*

- **Complementation:** $\forall a \in comp^{-}(C).\, \gamma(a) \cap C = \emptyset$*, and*
- **Maximal underapproximation:** $\forall a \in L.\, \gamma(a) \cap C = \emptyset \Rightarrow \exists a' \in comp^{-}(C) : a \preceq a'$

For some abstract domains $comp^{-}$ is an inexpensive operation. For example, in the interval domain its complexity is $O(|C|)$: $comp^{-}(\{2,7\}) = \{(-\infty, 1], [3,6], [8,\infty)\}$, and so on for larger sets. For other domains, however, $comp^{-}$ will be costly or even not computable. Section 5 discusses several domains, including a boolean predicate domain, where $comp^{-}$ causes Candidate Elimination to be non-feasible to compute.

Example 2. Using a concept language of intervals, we initialize $S = \bot$ and $G = \{\top\}$ and begin processing examples. The first example is $c = 0$, and $label(0)$ is negative. To handle a negative point, $comp^{-}(\{c\}) = \{(-\infty, -1], [1, \infty)\}$ is computed, then the new value of $G = \{\top \sqcap (-\infty, -1], \top \sqcap [1, \infty)\} = \{(-\infty, -1], [1, \infty)\}$. All members of G are equal or greater than S (and therefore consistent), so no filtering is required.

A second sample seen is $c' = 2$ and $label(2)$ is positive. To handle a positive sample, the algorithm computes $\beta(c') = [2, 2]$ and then computes the new value of $S = \bot \sqcup [2, 2] = [2, 2]$. We can now see that one of the members of G is no longer consistent with c', since if it were selected it would label c' as negative, which makes it incomparable with S, so it is filtered, yielding $G = \{[1, \infty]\}$.

Candidate Elimination is a general algorithm, which can be used both for active learning and offline learning from a given set of points. It has several active learning variations, including the CAL [8] and A^2 [4] algorithms for active version space concept learning. Since in active learning the algorithm itself selects the next point that will be labeled, these algorithms address the problem of selecting an advantageous next point. For this, they define the *region of disagreement*, as the set of all the points for which some two hypotheses that are currently viable disagree:

Definition 5 (Region of disagreement). *The* region of disagreement *(sometimes* region of uncertainty*) [8] for a version space $\mathcal{V}$ is $R_{\mathcal{V}} = \{c \in \mathcal{D} \mid \exists h_1, h_2 \in \mathcal{V} : h_1(c) \neq h_2(c)\}$.*

Selecting the next example from this region would guarantee the elimination of at least one hypothesis with every step.

The final result of candidate elimination would be one of the following: a single classifier $S = \bigsqcup_{c \in C^+} \beta(c)$, if S and G converge; no classifier, if S and G become inconsistent; or a (possibly infinite) range of classifiers described by the hypotheses bound between S and G, from which one or more can be selected.

3.2 Unbiased Learning

The concepts supported by the Candidate Elimination algorithms are conjunctive (specifically, cartesian concepts), and the need to find the next hypothesis that will be consistent with all examples while using only conjunction is the learner's *bias*. Bias is the way it generalizes about data it has not seen. However, as the following example demonstrates, for the case of programs, conjunctive concepts are not enough:

Example 3. Consider the differential analysis of `f(x)=x` and `g(x)=if (abs(x) < 1000) 0 else x` using the intervals concept language. These programs differ in intervals $[-1000, -1]$ and $[1, 1000]$, and are the same in $[MinInt, -1001]$, $[0, 0]$ and $[1001, MaxInt]$, so describing the difference (or similarity) using intervals requires disjunction. However, the (conjunctive) intervals language only allows to bound a set of points using a single interval. Thus, any concept describing all the positive points (where the programs agree) will also include negative points (where the programs disagree) and vice versa. Specifically, candidate elimination will finish as inconsistent if it sees a single negative sample amidst the positive samples.

Unbiased learning When disjunctions are added, more complex concepts can be described despite the limitation of the basic concept language L_0 (this is equatable to the powerset lattice over L_0). However, the added freedom that comes with disjunctions introduces a problem, which is inherent in the join operation of the powerset lattice: $a_1 \sqcup a_2 = a_1 \vee a_2$. If every specific example is generalized to $\beta(c)$ and then joined to the rest, the specific lower bound will never become more abstract than $\varphi_S = \bigvee_{c \in C^+} \beta(c)$. Similarly, if allowing arbitrary connectives, the generic upper bound will never become more refined than $\varphi_G = \bigwedge_{c \in C^-} \neg\beta(c)$.

This is what Mitchell calls "the futility of the unbiased learner" [25]. Once the ability to abstract is lost, the hypotheses at the bounds of the version space will never be able to answer yes or no about examples they have never seen, and unless the entire space is sampled (if this is at all possible), they will never converge.

3.3 Unbiased learning by partitioning the space

Mitchell's original work on version spaces [26] suggests handling an inconsistency that requires disjunction by working with a pre-partitioned space and performing the candidate elimination algorithm separately in every partition. While this approach is the most efficient, it requires prior knowledge of where the disjunctions are likely to occur, and a more flexible concept language that allows for the partition. Murray's tool HYDRA [27] uses an operation equivalent to $comp^{-}$ to dynamically partition the domain using the negative samples, creating regions where generalization is allowed. Every division of the space may cause a recalculation of impacted abstract elements, which need to be re-generalized within the newly created regions. In addition to requiring an efficient $comp^{-}$, HYDRA lacks a simple convergence condition, but rather is intended to run until either samples run out or the teacher is "satisfied" with the resulting description.

4 Learning Disjunctive Abstractions

In this section we describe our algorithm for learning disjunctive concepts. Just as in the Candidate Elimination algorithm, what we seek to find is a boolean function partitioning the input space into the positive and the negative sets, described using the concept language. As in Candidate Elimination, we are dependent on the assumption that this partition is expressible using the concept language. However, unlike Candidate Elimination, we consider a disjunctive concept language, $\mathcal{P}(L)$.

From here on, we interchangeably describe disjunctive concepts in $\mathcal{P}(L)$ as disjunctive formulas, e.g., $a_1 \vee a_2$, and as sets, e.g. $\{a_1, a_2\}$. Further, $\sqcup$ always denotes the join of L, as opposed to the join of $\mathcal{P}(L)$, which is simply disjunction or set union.

Our key idea is to combine the benefits of the generalization obtained by using the join of L, with the expressiveness allowed by disjunctions. We therefore define a *safe generalization* which generalizes a set of concepts (abstract elements) $A \in \mathcal{P}(L)$ in a way that keeps them separate from a concrete set of counterexamples.

Definition 6 (Safe generalization). *A* safe generalization *of a set of concepts $A \in \mathcal{P}(L)$ w.r.t. a concrete set of counterexamples $C_{cex} \subseteq \mathcal{D}$ is a set $SG(A, C_{cex}) \in \mathcal{P}(L)$ which satisfies the following requirements:*

1. **Abstraction:** $\forall a \in A.\, \exists a' \in SG(A, C_{cex}).\, a \preceq a'$
2. **Separation:** $\forall a \in SG(A, C_{cex}).\, \gamma(a) \cap C_{cex} = \emptyset$
3. **Precision:** $\forall a \in SG(A, C_{cex}).\, \exists A' \subseteq A.\, a = \bigsqcup A'$

We say that $SG(A, C_{cex})$ is maximal *if whenever $a \in L$ satisfies the separation and the precision requirements, there exists $a' \in SG(A, C_{cex})$ s.t. $a \preceq a'$.*

Note that the separation requirement is the same as the "complementation" requirement of $comp^{-}$. Unlike the join of L which is restricted to return a concept in L, $SG(A, C_{cex})$ returns a concept in $\mathcal{P}(L)$, and as such it can "refine" the result of join in case $\bigsqcup A$ does not satisfy the separation requirement. The precision requirement is guided by the intuition that each $a \in SG(A, C_{cex})$, which represents a disjunct in the learned disjunctive concept, should generalize in accordance with the generalization of L and not beyond. If any of the conditions cannot be met, then $SG(A, C_{cex})$ is undefined. However, if $\gamma(A)$ and C_{cex} are disjoint, then $SG(A, C_{cex})$ is always defined because it will, at worst, perform no generalization and will return A.

Using safe generalization, we can define the "safe abstractions" of two sets C^+, C^-: $\varphi_{pos} = SG(\{\beta(c) \mid c \in C^+\}, C^-)$, which characterizes the positive examples, or $\varphi_{neg} = SG(\{\beta(c) \mid c \in C^-\}, C^+)$, which characterizes the negative examples (provided that SG is defined for them).

The ideal solution If C^+ and C^- partition the *entire* space and SG computes *maximal* safe generalization, then φ_{pos} and φ_{neg} will be the optimal solutions, in the sense of providing concepts with largest disjuncts which correctly partition $\mathcal{D}$. Note that in the case that the classifier is expressible as a concept in L, the ideal solution is equivalent to the result of Candidate Elimination, which is simply $\bigsqcup\{\beta(c) \mid c \in C^+\}$.

Since this definition, while optimal, is both unfeasible (for an infinite domain) and requires SG, which like $comp^-$ may be very expensive to compute, we propose instead a greedy algorithm to approximate it by directing the sampling of points in C^+ and C^- and by implementing SG with a heuristic approximation of maximality.

Our algorithm, D^3, is presented in Algorithm 2, and described below.

Two levels of abstraction D^3 modifies the version space algorithms to keep four hypotheses, divided into two levels of abstraction.

In the first level of abstraction, $\varphi_S, \varphi_{\neg G} \in \mathcal{P}(L)$ are formula representations of the minimal overapproximation of the points that have actually been seen. φ_S corresponds to Candidate Elimination's S, computed over $\mathcal{P}(L)$, for which join is simply disjunction. In an effort to simplify and only deal with disjunction and not negation, instead of G which underapproximates $\mathcal{D} \setminus C^-$, we use $\varphi_{\neg G}$ that abstracts C^- directly. In the second level of abstraction, $\varphi_{pos}, \varphi_{neg} \in \mathcal{P}(L)$ are added. These are incremental computations of the definition above, which provide safe generalizations of φ_S w.r.t. the current C^-, and of $\varphi_{\neg G}$ w.r.t. the current C^+.

Technically, $\varphi_S = \bigvee_{c \in C^+} \beta(c)$ and $\varphi_{pos} = \bigvee \psi_i$ s.t. $\psi_i = \beta(c_{i_1}) \sqcup \cdots \sqcup \beta(c_{i_k})$ for some $\{c_{i_1}, \ldots, c_{i_k}\} \subseteq C^+$. It can be seen that $C^+ \subseteq \gamma(\varphi_S) \subseteq \gamma(\varphi_{pos})$. Further, both φ_S and φ_{pos} are consistent with all the examples seen (including negative ones). Dually for C^-, $\varphi_{\neg G}$ and φ_{neg}.

D^3 updates the formulas as follows. Every positive sample c that arrives is first added to φ_S, and then if it is not already described by φ_{pos}, φ_{pos} is updated to a safe generalization of $\varphi_{pos} \vee \beta(c)$. If φ_{neg} is inconsistent with c, then any disjunct $\psi_i \in \varphi_{neg}$ for which $\psi_i(c) = true$ is refined by collapsing it into its original set of points, abstracting them using β and re-generalizing while considering the new point. Unlike Candidate Elimination, D^3 is symmetrical for positive and negative samples, hence negative samples are handled dually.

D^3 converges when φ_{pos} and φ_{neg} constitute a partition of $\mathcal{D}$. This means that $\varphi_{pos} \equiv \neg\varphi_{neg}$. This requires that in terms of expressiveness, the partition can be described both positively and negatively.

Greedily computing safe generalization Like $comp^-$, computing SG naively is doubly-exponential. We therefore use a greedy strategy. SG first finds all the subsets of the input whose join is consistent. This is done inductively bottom-up, based on the fact that if $a_1 \sqcup a_2$ is inconsistent with some point c, then $a_1 \sqcup a_2 \sqcup a_3$ will be as well. This means the bottom-up construction can stop generalizing at smaller subsets. From the computed consistent generalized concepts, a coverage of the input is selected greedily using a *cardinality function*: $\mathcal{P}(L) \rightarrow \mathbb{N}$ that assigns a value to the desirability of a sub-

Algorithm 2: The D^3 algorithm

```
Input: $O$ oracle, $label$ teacher function
$\varphi_{pos} \leftarrow false$; $\varphi_{neg} \leftarrow false$
$\varphi_S \leftarrow false$; $\varphi_{\neg G} \leftarrow false$
$C^+ \leftarrow \emptyset$; $C^- \leftarrow \emptyset$
while $((\varphi_{pos} \vee \varphi_{neg} \not\equiv true) \vee (\varphi_{pos} \wedge \varphi_{neg} \not\equiv false)) \wedge \neg timeout$ do
    // Check for consistency
    $c_{pos} \leftarrow O\mid_{\varphi_S}$; $c_{neg} \leftarrow O\mid_{\varphi_{\neg G}}$
    if $label(c_{neg})$ is positive $\vee$ $label(c_{pos})$ is negative then
        return no classifier
    // Sample every region of disagreement
    $c_1 \in O\mid_{\neg\varphi_{pos} \wedge \neg\varphi_{neg}}$
    $c_2 \in O\mid_{\varphi_{pos} \wedge \varphi_{neg}}$
    $c_3 \in O\mid_{\varphi_{pos} \wedge \neg\varphi_S}$
    $c_4 \in O\mid_{\varphi_{neg} \wedge \neg\varphi_{\neg G}}$
    $C = \{c_1, c_2, c_3, c_4\}$
    for $c \leftarrow C$ do
        if $label(c)$ is positive then
            $\varphi_S \leftarrow \varphi_S \vee \beta(c)$
            if $\neg\varphi_{pos}(c)$ then $\varphi_{pos} \leftarrow$ SG$(\varphi_{pos} \vee \beta(c), C^-)$;
            if $\varphi_{neg}(c)$ then $\varphi_{neg} \leftarrow$ refine$(\varphi_{neg}, c, C^-, C^+)$;
            $C^+ \leftarrow C^+ \cup \{c\}$
        else // Symmetrical
            $\varphi_{\neg G} \leftarrow \varphi_{\neg G} \vee \beta(c)$
            if $\neg\varphi_{neg}(c)$ then $\varphi_{neg} \leftarrow$ SG$(\varphi_{neg} \vee \beta(c), C^+)$;
            if $\varphi_{pos}(c)$ then $\varphi_{pos} \leftarrow$ refine$(\varphi_{pos}, c, C^+, C^-)$;
            $C^- \leftarrow C^- \cup \{c\}$
return $\varphi_{pos}, \varphi_{neg}$
```

Function SG($\varphi = \psi_1 \vee \cdots \vee \psi_k, C_{cex}$)

```
$consistent \leftarrow \{\{\psi_j\} \mapsto \psi_j \mid 1 \leq j \leq k\}$ // $lvl = 1$
for $lvl \leftarrow 2 \ldots k$ do
    $prevLvl \leftarrow \{S \mid S \in \mathcal{P}(\{\psi_1, \ldots, \psi_k\}), |S| = lvl - 1, S \in dom(consistent)\}$
    $pairs \leftarrow \{(S, S') \mid S, S' \in prevLvl, |S \cup S'| = lvl\}$
    for $(S, S') \leftarrow pairs$ do
        // Can be optimized to not check the same $S \cup S'$ twice
        if $S \cup S' \notin dom(consistent)$ then
            $a \leftarrow consistent[S] \sqcup consistent[S']$
            if $\gamma(a) \cap C_{cex} = \emptyset$ then
                $consistent \leftarrow consistent \cup \{S \cup S' \mapsto a\}$
$seen \leftarrow \emptyset$; $res \leftarrow \emptyset$
while $seen \neq \{\psi_1, \ldots, \psi_k\}$ do
    $joint \leftarrow \arg\max_x\{cardinality(x) \mid x \in dom(consistent), x \cap seen = \emptyset\}$
    $seen \leftarrow seen \cup joint$
    $res \leftarrow res \cup consistent[joint]$
return $\bigvee res$
```

```
**Function** refine($\varphi = \psi_1 \vee \cdots \vee \psi_k, c, C_{abstracted}, C_{cex}$)
$contradicting \leftarrow \{\psi_i \mid \psi_i(c), 1 \leq i \leq k\}$
$consistent \leftarrow \{\psi_i \mid 1 \leq i \leq k\} \setminus contradicting$
for $\psi \leftarrow contradicting$ do
    $concrete \leftarrow \{c' \in C_{abstracted} \mid \psi(c')\}$
    $generalized \leftarrow \texttt{SG}(\bigvee\{\beta(c') \mid c' \in concrete\}, C_{cex})$
    $consistent \leftarrow \texttt{SG}(consistent \cup \{\theta_i \mid generalized = \theta_1 \vee \cdots \vee \theta_j\}, C_{cex})$
return $\bigvee consistent$
```

set of L to the coverage. A default cardinality function returns the number of concepts in the subset, preferring a generalized element created from the largest subset, but some domains allow for a better one. This greedy selection no longer ensures maximality.

If the domain is assured to be one for which $comp^{-}$ is efficient to compute, HYDRA's technique (Section 3.3) can be used to partition the space so that re-joining after a contradiction has been refined around is linear. While this is not always possible, the greedy computation of SG is improved to an exponential operation. Care is taken to always perform it on the fewest possible elements. With the exception of backtracking (calls to `refine` that do collapse a disjunct), calls to SG will encounter a set of elements most of which cannot be joined to each other, and the computation will never try computing any larger joins containing them.

Example 4. In sampling inputs for `f(x)` and `g(x)` from Example 3, using intervals as our concept language, consider the case where the algorithm has already seen the concrete points: $\{0, 1002, -837\}$ which means it has learned $\varphi_S = [0,0] \vee [1002, 1002]$ and $\varphi_{\neg G} = [-837, -837]$. (Recall that positive points correspond to program similarity and negative points correspond to a difference.) It has also generalized $\varphi_{pos} = [0, 1002]$, as right now it is a valid hypothesis that is not in contradiction with any data, and since there is nothing to abstract, $\varphi_{neg} = [-837, -837]$.

When the algorithm sees a new concrete point 478, for which $f(478) \neq g(478)$, it expands $\varphi_{\neg G}$ to include the point. It also adds a second clause so $\varphi_{neg} = [-837, -837] \vee [478, 478]$. It then tries to generalize this using the intervals lattice, where $[-837, -837] \sqcup [478, 478] = [-837, 478]$ but this new classifier is consistent with the fact that $f(0) = g(0)$ and $0 \in [-837, 478]$. This means these two points cannot be abstracted together. Likewise, φ_{pos} is tested, and since $478 \in [0, 1002]$ it has become inconsistent, so it is refined into $[0,0] \vee [1002, 1002]$.

We examine another point, 10004, where $f(10004) = g(10004)$, which is added to φ_S and then to φ_{pos}. φ_{pos} is now comprised of $[0,0] \vee [1002, 1002] \vee [10004, 10004]$. While $[0, 1002]$ and $[0, 10004]$ are inconsistent with what we know about 478, $[1002, 10004]$ is consistent, so we abstract $\varphi_{pos} = [0,0] \vee [1002, 10004]$.

It should be noted that while φ_S and $\varphi_{\neg G}$ advance in one direction, φ_{pos} and φ_{neg} travel up and down the lattice in order to stay consistent.

Regions of disagreement As shown in Section 2, the four formulas maintained by D^3 partition the region of disagreement (Definition 5) into four separate regions, which can be seen in Fig. 2. The *uncovered* and *covered disagreement* regions represent a classification disagreement between the two abstractions, and the *positive* and *negative abstracted disagreement* are a classification disagreement between the bounds and the abstractions. Since all four formulas are consistent with all previously processed points, these will all be unsampled regions of the space.

Sampling the uncovered or covered disagreement regions is guaranteed to advance the algorithm by either abstracting or refining at least one of the formulas. By sampling the region with an *oracle*, advantageous points can be selected. In the case of sampling $\varphi_{pos} \wedge \neg\varphi_S$ or $\varphi_{neg} \wedge \neg\varphi_{\neg G}$ (note that these formulas are not concepts in $\mathcal{P}(L)$, they are just used as an interface to the oracle), it is possible that while the sampled point will be added to φ_S or $\varphi_{\neg G}$, it will make no change in the generalized formulas, and in essence not advance the algorithm toward convergence.

Timeout and consistency checks Like Candidate Elimination, if D^3 recognizes that the concept language is not descriptive enough to allow it to converge, it returns "no classifier". This will happen if the abstraction inherent in β causes inconsistency. To test for this, the algorithm samples specifically for unseen points in φ_S and $\varphi_{\neg G}$ and if they indicate inconsistency, returns "no classifier". If β is precise enough, there will be no such unseen points, and this test will require no action.

Another option is that convergence is unattainable, either because the partition of the space cannot be described by $\mathcal{P}(L)$ from neither the positive nor the negative direction, which will cause a loop of φ_{pos} and φ_{neg} continuously generalizing to intersect each other and refining to create uncovered space, or because the domain is infinite and not advantageously sampled by the oracle. A timeout is introduced to stop the process. Our experiments have shown a timeout of 2000 iterations to be sufficient. In case of timeout, φ_{pos} and φ_{neg} can still be returned (as both are consistent with the seen points).

4.1 D^3 on a fixed example set

If a fixed set of examples C is given to D^3 along with a $label$ function defined only over C, which means there is no teacher to query about new samples, the convergence condition no longer applies. D^3 will run until samples are exhausted, and the role of the oracle will no longer be to provide a new sample, but rather to order the samples so that the algorithm will have to do as little backtracking as possible, and will be more efficient than simply computing SG.

For example, for the intervals domain, the oracle would return the samples in ascending order, which would ensure no counterexample dividing an interval disjunct would ever be provided.

5 Prototype Implementation and Evaluation

We have implemented the D^3 algorithm in Scala for several domains, along with several input sampling strategies (oracles). In this section we first describe the sampling strategies and concept languages implemented in our differential analysis experiments. We then describe our experimental results over a small but challenging set of programs.

5.1 Input Sampling Strategy

An ideal oracle would offer points that would lead D^3 to finding every disjunct in the desired φ_{pos} and φ_{neg}, and that would lead to convergence. It would also order the samples so that the number of refinements would always be minimal, and the algorithm would converge on a precise result using the fewest operations (However, note that the result of D^3 is not sensitive to order of the sampled points).

Coming up with an ideal oracle is in general undecidable. Instead, one may choose between different heuristics for discovering interesting input values.

Naively, a requested region is sampled uniformly. However, this often misses singularity points (e.g., due to testing `if (x != 0)`). A slightly better approach is to use biased sampling with tools such as ScalaCheck [1] that favor "problematic values" such as 0, -1, or `Int.MaxValue`. Other techniques, typically used to increase test coverage (e.g., concolic testing [33], whitebox fuzzing [16]), can be applied here as well.

Another practical solution is to use a *grey-box* approach. For instance, searching the code for constants and operations, and generating from them a list of special values that should be returned to the algorithm when they match one of the sampled regions. We have implemented a constants-only oracle which has proved sufficient for all implemented numerical domains.

5.2 Intervals and Intervals with Congruence

Intervals We use a standard intervals lattice [9] with $|\gamma([l,h])|$ as a cardinality measure for an interval $[l,h]$. This measure is easily computed and directs the greedy choice towards the largest intervals.

D^3 with the intervals domain has the property that if some point from a positive or negative region is seen, the algorithm will not converge until the entire region is discovered. This is because $\bigsqcup(\{\beta(c) \mid c \in C\}) = [l,h]$ only if $l, h \in C$, and since in order for D^3 to converge, the space needs to be covered (i.e. $l-1$ and $h+1$ are, themselves, described by φ_{pos} or φ_{neg}), both sides of every boundary between φ_{pos} and φ_{neg} are sampled. This means that the grey-box oracle would be adequate because relevant points are likely to be present as constants in the code.

While intervals are useful for some examples (see Tab. 1), they cannot handle examples such as that in Fig. 1. Running D^3 with intervals on Fig. 1 and assuming a finite domain of 32-bit integers will only converge by sampling the whole domain. While a full description of similarity ($\{[x,x] \mid x \mod 10 = 0 \wedge -2147483648 \leq x \leq 2147483647\}$ for a 32-bit integer) exists, it consists of 400 million separate disjuncts. And since these disjuncts contain one concrete element each, they are also likely to never be discovered and instead be overapproximated by the description of difference. What the interval concept language lacks is the ability to abstract these into one description.

Intervals with congruence We consider a richer domain of intervals with congruences for several divisors. Instead of using the full congruence abstract domain [17], we use its collapsed versions to the divisors 2 through 10 that allow the information on several different congruences to be preserved simultaneously. This allows us to learn the similarity $(x \leq 2147483640 \wedge x \geq -2147483640 \wedge x \mod 2 = 0 \wedge x \mod 5 = 0 \wedge x \mod 10 = 0) \vee (x \leq 9 \wedge x \geq -9)$ for the example of Fig. 1.

Like intervals, the cardinality measure for intervals with congruence counts the number of elements in the interval, accounting for all congruences that are not $\top$. Using the grey-box oracle, the Sum of digits example converges with the expected difference of $\bigvee_{i \in 1 \ldots 9} (x \leq -1 \wedge x \mod 10 = i) \vee (x \geq 11 \wedge x \mod 10 = i)$.

Larger arities Both the intervals and intervals with congruence domains can be applied to functions of different arities by using the product domain for as many arguments to the function as necessary. We have implemented the domain $Intervals \times Intervals$ for functions that take two `int` parameters.

Using the same grey-box oracle lifted to the product domain for two parameters, and the area of the box as the cardinality function, we tested D^3 on several samples including the `Quadrant` test, in which the exercise is to take a point in the geometric plane (x, y) and return the quadrant it is in. One implementation defines Quadrant I as $x > 0, y > 0$ and the other as $x \geq 0, y \geq 0$, and the same for Quadrant II and IV.

This yields the difference of $x = 0 \vee y = 0$, and similarity of $(x \geq 1 \wedge y \geq 1) \vee (x \geq 1 \wedge y \leq -1) \vee (x \leq -1 \wedge y \geq 1) \vee (x \leq -1 \wedge y \leq -1)$.

5.3 Quantified boolean predicates over arrays

In the domain of quantified boolean predicates over arrays, $comp^-$ causes the number of upper bounds to grow exponentially, which means Candidate Elimination is non feasible to compute, even for simple conjunctive descriptions. D^3 finds these descriptions, as well as disjunctive ones.

Creating predicates Since we have no property or assertions from which to draw predicates, we use a template-based abstraction, as in [21, 36, 34]. For simplicity, we use a fixed set of predicate templates filtered by correctness on the concrete array, similar to those used by the Houdini [14] and Daikon [12] annotation assistants.

The β function In our implementation $\beta(c)$ is a conjunction of all the facts the templates discover about it. For very small arrays we can allow a precise beta function that generates specific predicates for $arr(0), arr(1), \ldots$. For larger arrays we keep more compact facts such as: $\forall i.(arr(i) \leq arrMax \wedge arr(i) \geq arrMin)$ for the maximum and minimum values in the array. This is not a precise β, which illustrates the importance of the consistency check in Section 4.

Oracle The grey-box oracle approach, which works for most integer functions, is insufficient for arrays - the simple syntactic analysis of the code is insufficient for inferring meaningful examples. To demonstrate the D^3 algorithm, we provide our experiments with a manual "oracle procedure" specific to the test.

The `Find2` test finds the occurrence of 2 in an array without using array functions. The spec implementation provided is simply `arr.indexOf(2)`, and the tested implementation makes an off-by-one error failing to test `arr(0)`. D^3 learns the difference $arr(0) \geq 2 \wedge arr(0) \leq 2$.

5.4 Experimental Evaluation

Tab. 1 compares each of the tests to the capabilities of a conjunctive method such as joining all the samples of a large set of positive examples or running an active version of Candidate Elimination. The columns for "conjunctive (difference)" and "conjunctive (similarity)" signify whether the analysis would succeed if performed when treating the different points or the similar points as C^+.

Tests `Example 3`, `Sum of Digits`, `Quadrant` and `Find2` have been discussed in detail previously. `Square` tests the difference between two implementations of squaring a number, one which casts to Long, and another that does not, thus causing an integer

	test name	conjunctive (difference)	conjunctive (similarity)	D^3
Intervals	Example 3	✗	✗	✓
	Square	✗	✓	✓
	StringAtIndex	✗	✗	✓
Intervals with congruence	IsOdd	✗	✓	✓
	Sum of Digits	✗	✗	✓
Boxes	SolveLinEq	✓	✗	✓
	Quadrant	✗	✗	✓
Boolean predicates over arrays	Find2	✗	✗	✓
	ArrayAverage	✗	✗	✓
	ArrayMaximum	✗	✗	✓

Table 1: Comparing the D^3 algorithm to the capabilities of conjunctive algorithms

overflow, creating a difference in any number that is large enough or small enough so that its square does not fit into an Integer. `StringAtIndex` returns the character at the given index of a string, or null if the index is outside the bounds of the string, where one implementation has an off-by-one error, causing the 0th place in the string to be returned as null and an exception to be thrown at one past the end. `IsOdd` tests a parameter for oddness, where one implementation incorrectly tests negative numbers. `SolveLinEq` returns the solution to a linear equation $ax^2 + b = 0$ where a, b are the arguments of the functions. One implementation is undefined where $a = 0$ and the other only when both a and b are zero. `ArrayAverage` averages the values in an array, where one implementation is undefined (division by zero error) when the array is empty and the other returns zero. `ArrayMaximum` searches for the maximum value in an array, where one implementation has incorrect initialization of the maximum to 0 rather than the first element, thereby returning an incorrect result when $\forall i : arr(i) < 0$.

As Tab. 1 shows, while some cases can be solved by attempting a conjunctive technique from both directions separately and taking one if its result has not become inconsistent, this will not work for others. In addition, in domains like predicates on arrays where $comp^-$ is not available, the lowering of the upper bound is not available, so only more primitive techniques such as generating a large number of samples with no guidance and attempting their join remain.

6 Related Work

Disjunctive abstraction Since the original work introducing the powerset construction for adding disjunction to a domain [10], there have been attempts to create a more practical abstraction that allows for disjunction but also for the abstraction that is limited by the powerset domain's join operation, as discussed in Section 3.2. The easiest limitation which introduces bias is limiting the number of allowed disjuncts [23, 30, 6]. While this forces an abstraction once the number of disjoint elements is reached, it may still be forced to cover a negative example because the number of elements allowed is insufficient. Another is the *finite powerset domain* [2, 3] which keeps only the finite sets in the powerset. However, this still retains the problem of the non-abstracting join.

The Boxes domain [20], based decision diagrams, is compact in representation even for a large number of disjuncted boxes. It is specialized for the $Integer \times Integer$ domain, though the technique might be extendible to other domains. Donut domains [15] allow for "holes" in a single convex set. This does not allow a disjunction of the positive sets, and cannot be used with non-convex domains such as congruence.

Disjunctive approaches to Candidate Elimination Several methods have been mentioned in Section 3.3. In [22], every step of the algorithm maintains n representations of the specific and general bounds from 1 to n disjuncts, where n is the number of samples seen. Then, the desired option can be selected either by convergence or by other criteria. This method is both wasteful in representation if many samples are required to cover the space, and the criteria for the best disjunction update are complex and require additional learning algorithms. Sebag's [31] approach learns a disjunction of conjunctions of negative examples. This is meant to cope with noisy datasets, which are irrelevant in the case of performing an abstraction, and decides at classification time based on tuning, which means its output is not a description of the space but rather a function capable of labeling individual points.

Abstracting with learning algorithms Thakur et al. [37] use a variation on Candidate Elimination to compute symbolic abstraction. Gupta et al. [19] present an algorithm for actively learning an automaton that separates the language of two models, called the separating automaton. Like D^3, this is an active learning algorithm based on asking a teacher to compute language inclusion. However, this algorithm is relevant only to string languages (models and automata). Counterexample-driven abstraction refinement techniques [5, 7, 18] for verification behave like a learning algorithm, requesting "classification" of a point or a trace, and eliminating it from the abstract representation, in much the same way as Candidate Elimination and D^3 do.

7 Conclusion

We presented D^3, an active learning algorithm for computing an abstraction of a set of positive examples, separating them from a set of negative examples. A critical component of D^3 is the *safe generalization* operation which transforms an element in a powerset domain into a more general element that does not intersect any negative point. In cases where D^3 can actively query additional points beyond an initial set, it aims at learning a partition of the entire space (and not only abstract the initial samples). We apply D^3 to compute an abstract semantic difference between programs using several abstract domains. We show that D^3 can compute a precise description of difference/similarity for several small but challenging examples.

Acknowledgment

The research leading to these results has received funding from the European Union's - Seventh Framework Programme (FP7) under grant agreement no. 615688 - ERC-COG-PRIME and under ERC grant agreement no. 321174-VSSC, and from the BSF grant no. 2012259.

References

1. Scalacheck: Property-based testing for scala.

2. Bagnara, R. A hierarchy of constraint systems for data-flow analysis of constraint logic-based languages. *Science of Computer Programming 30*, 1 (1998), 119–155.
3. Bagnara, R., Hill, P. M., and Zaffanella, E. Widening operators for powerset domains. *STTT 8*, 4-5 (2006), 449–466.
4. Balcan, M.-F., Beygelzimer, A., and Langford, J. Agnostic active learning. In *Proceedings of the 23rd international conference on Machine learning* (2006), ACM, pp. 65–72.
5. Beckman, N. E., Nori, A. V., Rajamani, S. K., Simmons, R. J., Tetali, S. D., and Thakur, A. V. Proofs from tests. *Software Engineering, IEEE Transactions on 36*, 4 (2010), 495–508.
6. Beyer, D., Henzinger, T. A., and Théoduloz, G. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification* (2007), Springer, pp. 504–518.
7. Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement. In *Computer aided verification* (2000), Springer, pp. 154–169.
8. Cohn, D., Atlas, L., and Ladner, R. Improving generalization with active learning. *Machine learning 15*, 2 (1994), 201–221.
9. Cousot, P., and Cousot, R. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming* (1976), Dunod, Paris, France, pp. 106–130.
10. Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1979), ACM, pp. 269–282.
11. Cousot, P., and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In *POPL* (1978), pp. 84–96.
12. Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on 27*, 2 (2001), 99–123.
13. Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming 69*, 1 (2007), 35–45.
14. Flanagan, C., and Leino, K. R. M. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity* (London, UK, UK, 2001), FME '01, Springer-Verlag, pp. 500–517.
15. Ghorbal, K., Ivančić, F., Balakrishnan, G., Maeda, N., and Gupta, A. Donut domains: Efficient non-convex domains for abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation* (2012), Springer, pp. 235–250.
16. Godefroid, P., Levin, M. Y., and Molnar, D. Sage: whitebox fuzzing for security testing. *Queue 10*, 1 (2012), 20.
17. Granger, P. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics 30*, 3-4 (1989), 165–190.
18. Gulavani, B. S., Chakraborty, S., Nori, A. V., and Rajamani, S. K. Automatically refining abstract interpretations. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 443–458.
19. Gupta, A., McMillan, K. L., and Fu, Z. Automated assumption generation for compositional verification. In *Computer Aided Verification* (2007), Springer, pp. 420–432.
20. Gurfinkel, A., and Chaki, S. Boxes: A symbolic abstract domain of boxes. In *Static Analysis*. Springer, 2010, pp. 287–303.

21. LOPES, N. P., AND MONTEIRO, J. Weakest precondition synthesis for compiler optimizations. In *Verification, Model Checking, and Abstract Interpretation* (2014), Springer, pp. 203–221.
22. MANAGO, M., AND BLYTHE, J. Learning disjunctive concepts. In *Knowledge representation and organization in machine learning*. Springer, 1989, pp. 211–230.
23. MAUBORGNE, L., AND RIVAL, X. Trace partitioning in abstract interpretation based static analyzers. In *Programming Languages and Systems*. Springer, 2005, pp. 5–20.
24. MINÉ, A. The octagon abstract domain. *Higher-Order and Symbolic Computation 19*, 1 (2006), 31–100.
25. MITCHELL, T. *Machine Learning*. McGraw-Hill international editions - computer science series. McGraw-Hill Education, 1997, ch. 2, pp. 20–51.
26. MITCHELL, T. M. *Version spaces: an approach to concept learning.* PhD thesis, Stanford University, Dec 1978.
27. MURRAY, K. S. Multiple convergence: An approach to disjunctive concept acquisition. In *IJCAI* (1987), Citeseer, pp. 297–300.
28. PARTUSH, N., AND YAHAV, E. Abstract semantic differencing for numerical programs. In *SAS* (2013), pp. 238–258.
29. PARTUSH, N., AND YAHAV, E. Abstract semantic differencing via speculative correlation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications* (New York, NY, USA, 2014), OOPSLA '14, ACM, pp. 811–828.
30. SANKARANARAYANAN, S., IVANČIĆ, F., SHLYAKHTER, I., AND GUPTA, A. Static analysis in disjunctive numerical domains. In *Static Analysis*. Springer, 2006, pp. 3–17.
31. SEBAG, M. Delaying the choice of bias: A disjunctive version space approach. In *ICML* (1996), Citeseer, pp. 444–452.
32. SEGHIR, M. N., AND KROENING, D. Counterexample-guided precondition inference. In *Programming Languages and Systems*. Springer, 2013, pp. 451–471.
33. SEN, K., AND AGHA, G. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification* (2006), Springer, pp. 419–423.
34. SHARMA, R., AND AIKEN, A. From invariant checking to invariant inference using randomized search. In *Computer Aided Verification* (2014), Springer, pp. 88–105.
35. SHARMA, R., SCHKUFZA, E., CHURCHILL, B. R., AND AIKEN, A. Data-driven equivalence checking. In *OOPSLA* (2013), pp. 391–406.
36. SRIVASTAVA, S., AND GULWANI, S. Program verification using templates over predicate abstraction. In *ACM Sigplan Notices* (2009), vol. 44, ACM, pp. 223–234.
37. THAKUR, A., ELDER, M., AND REPS, T. Bilateral algorithms for symbolic abstraction. In *Static Analysis*. Springer, 2012, pp. 111–128.