Automating and Validating Program Annotations

Mark Grechanik, Kathryn S. McKinley, and Dewayne E. Perry
Department of Computer Sciences
The University of Texas at Austin

Abstract

Program annotations help to catch errors, improve program understanding, and specify invariants. Adding annotations, however, is often a manual, laborious, tedious, and error prone process especially when programs are large. We offer a novel approach for automating a part of this process. Developers first specify an initial set of annotations for a few variables and types. Our *LEearning ANnnotations (Lean)* system combines these annotations with run-time monitoring, program analysis, and machine-learning approaches to discover and validate these annotations on unannotated variables. We evaluate our prototype implementation on open-source software projects and our results suggest that a modest set of annotations capture many program variables, and Lean can generalize from a small set of annotated variables to annotate many other variables. After users annotate approximately 6% of the program variables and types, Lean correctly annotates an additional 69% of variables in the best case, 47% in the average, and 12% in the worst case.

1. Introduction

Program annotations assert facts about programs and add them to the source code as comments or with language support. For example, an annotation may assert that values of program variables stay within certain ranges. Annotations may be used to describe program types, values, or identifiers. Program annotations help to catch errors, improve program understanding, recover software architecture, and specify invariants. One of the major uses of program annotations is to help programmers understand legacy systems. A Bell Labs study shows that up to 80% of programmer's time is spent discovering the meaning of legacy code when trying to evolve it [1]. Thus, the extra work required to annotate programs is likely to reduce development and maintenance time, as well as to improve software quality. However, annotating programs is often a manual, tedious, and error prone process especially for large programs. Although some programming languages (e.g., C#, Java) have support for annotations, many programmers do not sufficiently annotate or do not annotate their code at all.

Often programmers lack full knowledge of the source code to be capable of good annotations. Curtis' law [2] states that application and domain knowledge is thinly spread, and at most one or two team members may possess the full knowledge of a software system. Poor annotation quality is a result of this law since it is combined with the difficulty of mapping semantic concepts onto source code. A fundamental question for creating more robust and extensible software is how to annotate program source code with a high degree of automation and precision.

In this work, we focus on deriving semantic concepts for unannotated variables from an initial set of annotated variables. Simply stated, *semantic concept annotations* are nouns with well-accepted meanings in public or domain-specific knowledge. For example, the noun `Address` is a semantic concept meaning a place where a person or institution is located. Programmers may introduce variables named `Address`, `Add`, or `S[1]`, all for the `Address` concept[1]. The name of the variable `S[1]` does not match `Address`, and relating this variable to the `Address` concept is challenging. While the variable named `Add` partially matches `Address`, it is ambiguous if the program also uses a `Summation` concept for adding numbers.

Our solution, called *LEarning ANnotations (Lean)*, combines program analysis, run-time monitoring, and machine learning to automatically apply a small set of initial semantic annotations to additional unannotated types and variables. The input to Lean is program source code and a concept diagram describing relations between semantic concepts. The core idea of Lean is that after programmers provide a few initial annotations of some variables and types with these semantic concepts, the system will glean enough information from these annotations to annotate much the rest of the program automatically. We define annotation rules that guide the assignment of semantic concepts to program entities, and resolve conflicts when they arise.

[1] These names are taken from open-source program *Vehicle Maintenance Tracker* whose code fragments are shown in Figure 1.

Lean works as follows. After programmers specify initial annotations, Lean instruments a program to perform run-time monitoring of program variables. Lean executes this program and collects a profile of the values of instrumented variables. Lean uses this profile to train its learners to identify variables with similar profiles. Lean's learners then classify the rest of program variables by matching them with the semantic concept annotations. Once a match is determined for a variable, Lean annotates it with the matching semantic concept.

Annotating 100% of variables automatically is not realistic. Many reasons exists: machine learning approaches do not guarantee 100% success in solving problems; concept diagrams representing program design specifications may not match the entire program; and some concepts may be difficult to relate to program variables due to lack of modularity. Consequently, Lean makes some mistakes when learning annotations. In order to improve its precision, Lean uses relations between variables and their corresponding concepts to validate learnt annotations. Program analysis determines relations among annotated variables, and these relations are compared with corresponding relations in concept diagrams. If a relation is presented between two variables in the program code and there is no relation between concepts with which these variables are annotated, then Lean flags it as a possible annotation error.

We evaluate our approach on open-source software projects and obtain results that suggest it is effective. Our results show that after users annotate approximately 6% of the program variables and types, Lean correctly annotates an additional 69% of variables in the best case, 47% in the average, and 12% in the worst case.

2. A Motivating Example

The *Vehicle Maintenance Tracker (VMT)* is an open source Java application that records the maintenance of vehicles (e.g., boats, planes, or cars) [3]. Fragments of the VMT code from three different files are shown in Figure 1a-c, and a concept diagram is shown in Figure 1d.

A fragment of code from the file `vendors.java` shown in Figure 1a contains the declaration of the class vendors whose member variables of type `String` are `Name`, `Add`, `Pho`, `Email`, and `Web`. These variables stand for the vendor's name, address,

```
public class vendors {
  private String Name,  Add,   Pho, Email, Web;
              …………….
}
```

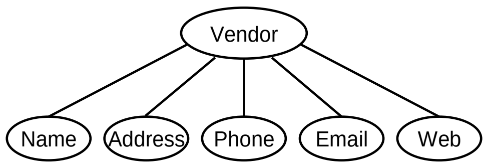
a) File vendors.java

```
public class VendorEdit extends InternalFrame {
            private Text NameText;
            private TextArea AddressText;
            private TextArea PhoneText;
            private Text EmailText;
            private Text WebText; }
```

b) File VendorEdit.java

```
public void addMaintenanceEditor(String[] S) {
addMaintenanceServices(new String[]{
   ((MaintenanceEdit)Desktop.getSelectedFrame()).
            getName(),  S[4], S[5]});
   }
};
String s = S[1];
if (s.equalsIgnoreCase(""))
    s = "New";
String residence = S[3];
```

c) File VMT.java



d) A concept diagram for the VMT application

Figure 1. Code fragments from programs of the VMT project and its concept diagram.

phone number, email, and web page concepts respectively. A fragment of the code from the file `VendorEdit.java` shown in Figure 1b contains the declaration of the class `VendorEdit` whose member variables of types `Text` and `TextArea` represent the same concepts. Even though the names of these variables in the class `VendorEdit` are different from the names of the corresponding variables in the class `vendors`, their names partially match. For example, the variable name `Pho` in the class `vendors` matches the variable `PhoneText` in the class `VendorEdit` more than any other variable of this class when counting the number of consecutive letters matched.

This matching procedure does not work for the fragment of code shown in Figure 1c. To what semantic concept does the variable `S`, which is the parameter to the method `addMaintenanceEditor`, correspond? It turns out that the variable `S` is an array of `Strings`, and its elements `S[1]`, `S[2]`, `S[3]`, `S[4]`, and `S[5]` hold values of vendor’s identifier, address, email, phone number, and web page concepts respectively. No VMT documentation mentions this information, and programmers have to run the program and observe the values of these variables in order to discover their meanings.

Lean can automate the process of annotating classes and variables shown in Figure 1a-c with concepts from the diagram shown in Figure 1d. This diagram is a graph whose nodes designate semantic concepts and edges specify relations among these concepts. We observe that spelling of some variable names are similar to the names of corresponding concepts, i.e., `Pho` – `Phone`, `Add` – `Address`, `Web` – `WebSite`, `Name` – `Name`, and `Email` – `Email`. Specifically, parts of the names of the variables are contained within the names of the concepts and vice versa. Lean uses these similarities match names of variables and concepts, and subsequently annotate variables with semantic concepts.

Variables `residence` and `Address` are spelled differently, but they are synonyms. Extended with a vocabulary linking synonymic words, Lean hypothesizes about similarities between words that are spelled differently but have the same meaning. These vocabularies can link domain-specific concepts used by different programmers thereby establishing common meanings for different programs. For example, "chip" and "dice" mean "microprocessor" in semiconductor manufacturing. If the name of some variables contain these words, they are likely to correspond to the `microprocessor` concept.

By observing patterns in values of program variables Lean can determine whether they should be annotated with certain concepts. To observe patterns, Lean instruments source code to collect run-time values of the program variables. After running the instrumented program, Lean creates a table containing sample data for each variable. A sample table for the VMT application is shown in Table 1. Each column in this table contains variable name and values it held. Some values have distinct structures. The variable `Pho` contains only numbers and dashes in the format `xxx-xxx-xxxx`, where `x` stands for a digit and the dash is a separator. The variable `Name` contains only alphabetic characters. Values held by the variable `Email` have a distinct structure with the `@` symbol and dots used as separators. Lean learns the structures of values for annotated variables using machine-learning algorithms, and it then assigns the appropriate semantic concepts to variables whose values match the learnt structures.

Lean can also exploit statistical information with additional rules. For example, if words like `Circle`, `Drive`, `Road` occur in values, then Lean learners can classify this variable as the `Address` concept.

Email	Pho	Name	Web	Add
tc@abc.com	512-342-8434	John Smith	http://www.utexas.edu/~john	Tamara Circle, Austin
mcn@jump.net	512-232-3432	Mark Grechanik	http://www.utexas.edu/~mark	McNeil Drive, Austin
sims@su.edu	512-232-6453	John Perry	http://www.utexas.edu/~perry	Sims Road, Dallas
lg@ibm.com	512-877-3254	Mark Holtz	http://www.utexas.edu/~holtz	Laguna Hwy, Dallas

Table 1. Sample values taken by the program variables.

We also observe that if concepts are related in a diagram, then types and variables that are annotated with these concepts are related in the code too. The relation between concepts in a diagram means that instances of data described by these concepts are linked in some way. For example, the concept `Name` is related in the concept `Vendor`, in the concept diagram shown in Figure 1d. This relation can be expressed as "Vendor has a Name." The variable `Name` which is annotated with the concept `Name` is contained by the class `vendors` which is annotated with the concept `Vendor`. The containment relation in the source code corresponds to the "has-a" relation in the concept diagram. Lean explores program source code to analyze relations between program variables and types, and then compares them with relations among corresponding concepts in diagrams in order to infer and validate annotations. We explain this process in Section X.

3. The Problem Statement

This section defines the problem of automating and validating program annotation and validation formally. We define the structure of concept diagrams and types of relations between concepts in diagrams and variables and types in program code. Then, we present rules for annotating program variables and types, and give a formal definition of the problem.

3.1. Definitions

Feature modeling is a technique for modeling software with feature diagrams, where a *feature* is an end-user-visible characteristic of a system describing a unit of functionality relevant to some stakeholder [4][5]. For example, for the code shown in Figure 1 features are `Vendor`, `Name`, `Email`, `Phone`, `Website`, and `Address`. Concept diagrams used in feature modeling are called *feature diagrams (FD)*. Feature diagrams are graphs whose nodes designate features and edges (also called *variation points*) specify how two features are attached to each other.

Four basic types of diagrams are shown in Figure 2. Features f_1 and f_2 are *optional* if one of them or both or none can be attached to the base feature P. *Mandatory* are features f_3 and f_4 since both of them should be attached to the base feature P. Features f_5 and f_6 are *alternative* if either f_5 or f_6 are attached to the base feature P. Finally, an *or-feature* diagram specifies that either feature f_7 or feature f_8 and either feature f_9 or feature f_{10} are attached to the base feature P.

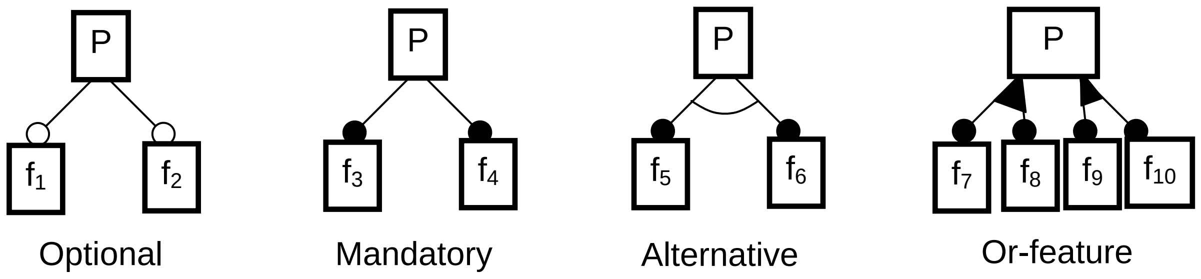


Figure 2. Basic types of feature diagrams.

Programmers start the annotation process with Java source code and a feature diagram which they use to specify initial mappings between features and a subset of program types and variables. This mapping is formalized in the definition of the Annotation Function.

Definition: Annotation Function. The annotation function α_o: $o \rightarrow 2^F$ maps a program object (variable) to a subset of features in a feature diagram F, and the annotation function α_t: $\tau \rightarrow 2^F$ maps a type (basic type, class, interface, or method) to a subset of features.

The dependency between a type and an object is expressed by the function Type(o) = τ, which maps the object o to its type $\tau \in T$, where T is the set of types. Without the loss of generality we use the annotation function α: $\pi \rightarrow 2^F$ that maps a program entity π (i.e., variable or type) to a subset of features in some feature diagram, where program entity $\pi \in \{\tau, o\}$.

Definition: Expression. The n-ary relation Expression $\subseteq v_1 \times v_2 ... \times v_n$ specifies an expression, where $v_1, v_2, ..., v_n$ are variables or methods used in this expression.

Definition: Navigate. The navigation relation Navigate(p,q) $\subseteq$ Expression(p, q) is the expression p.q where q is a member (e.g., field or method) of object p.

Definition: Assign. The assignment relation `Assign` $\subseteq$ `p` $\times$ `Expression` specifies the assignment expression `p = Expression`, where `p` is the variable on the left-hand side and `Expression` relation stands for some expression on the right-hand side.

Definition: Cast. The cast relation `Cast(p, q)` $\subseteq$ `Expression(p, q)` is the cast expression `(q)p` where `p` $\in$ `o` and `q` $\in$ `T` i.e., casting an object `p` to some type `q`.

Definition: Subtype. The relation `SubType(p, q)` specifies that a type `p` is a subtype of some type `q`. In Java this function is implemented via the implement interface clause. That is, a class `p` that implements some interface `q` is related to this interface via the `SubType` relation.

Definition: Inherit. The `Inherit(p, q)` relation specifies that a type `p` inherits (or extends in Java terminology) some type `q`.

Definition: Contains. The `Contains(p, q)` relation specifies that type or object `p` is contained within the scope of some type `q`. We call interfaces, classes, and methods containment types because they contain other types, fields and variables as part of their definitions. That is, interfaces contain method declarations, classes contain definitions of fields and methods, and methods contain uses of fields and declarations and uses of local variables that are instances of some types.

Definition: δ-relation. The relation $\delta(\pi_k, \pi_n)$ stands for "programming entity π_k is used in the same expression with programming entity π_n." For example, if two variables `p` and `q` are related via the `Expression`, `Navigate`, `Assign`, or `Cast` relations, then these variables are also related via the δ-relation, δ`(p, q)`.

This relation is irreflexive, antisymmetric, and transitive. For example, from the expression `x = y + z` we can build four δ relations: δ`(x, y)`, δ`(x, z)`, δ`(y, z)`, and δ`(z, y)`. If variables are used in the same expression and their values are not changed after this expression is evaluated, then their order is not relevant in the δ-relation. However, if the value of a variable is changed as a result of the evaluation of the expression, then this variable is the first component of the corresponding δ-relation.

Definition: γ-relation. The relation $\gamma(f_p, f_q)$ stands for "feature f_p is connected to feature f_q via a variation point."

This relation is irreflexive, antisymmetric, and transitive. For example, from the mandatory feature diagram shown in Figure 2, we can build two γ relations: $\gamma(P, f_3)$ and $\gamma(P, f_4)$. If features are not connected by a variation point, then their order is not relevant in the γ-relation. However, if a feature f_r is attached to a feature f_s, then feature f_s is the first component of the corresponding γ relation.

The function α maps pairs from the relation δ to pairs from the relation γ. Suppose that $(a, b) \in \delta$ and $(c, d) \in \gamma$. Then the element (a, b) is annotated with the element (c, d) if and only if $c \in \alpha(a)$ and $d \in \alpha(b)$. As a shorthand we write $(c, d) \in \alpha((a, b))$.

3.2. Rules of Program Annotations

When programmers map types (i.e., interfaces, classes, and methods) and their variables (i.e., fields and objects) to features, these mappings may conflict with one another. For example, if a class is mapped to one feature and it implements an interface that is mapped to a different feature, then what default mapping would be valid for an instance of this class? This section offers heuristic rules to resolve ambiguities when annotating programs.

Direct mapping: in general, we write $\gamma \in \alpha(\delta)$ to express the fact that for a δ-relation between objects in the source code that are annotated with feature labels there is a corresponding γ-relation between the corresponding features in some feature diagram.

Entity mapping: types and variables can be mapped to a set of features in the feature diagram. This rule is defined by the function $\alpha: \pi \rightarrow 2^F$, where F is a set of features. When a type is mapped to some feature, this type bears the label that is the name of the feature. Instances of this type are automatically annotated with its feature label. We write this rule as $Type(o) = \tau \wedge f \in \alpha(\tau) \rightarrow f \in \alpha(o)$. If the container type is mapped to some feature, then all of its members are automatically mapped to the same feature, i.e., $Contain(p, q) \wedge f \in \alpha(q) \rightarrow f \in \alpha(p)$.

Expression annotation: if variables in an expression defined by the relation $Expression(v_1, \ldots, v_n)$ are annotated with some set of features, that is, without the loss of generality $f_1 \in \alpha(v_1), \ldots, f_n \in \alpha(v_n)$, then Lean annotates it with a set of features as $f_1 \cup f_2 \cup \ldots \cup f_n$.

Assignment annotation: Given the relation `Assign(p, expr)`, the expression `expr` is annotated with a set of features `f`, then the variable `p` is annotated with the same set of features. The converse holds true, i.e., $\texttt{Assign(p, expr)} \wedge (\texttt{f} \in \alpha(\texttt{p}) \Leftrightarrow \texttt{f} \in \alpha(\texttt{expr}))$. For example, the variable `s` in the fragment of code shown in Figure 1c is assigned the value of the variable `S[2]`. This variable is mapped to the concepts `Address`. According to the assignment rule the variable `s` maps to the concept `Address`.

Cast annotation: casting an object `p` to some type `q` automatically remaps this object `p` to the feature to which this type is mapped. If the type `q` is not mapped to any feature, then the original mapping of the object `p` is not changed. That is, $\mathrm{Cast}(p, q) \wedge \alpha(q) = f \rightarrow \alpha(p) = f$ and $\mathrm{Cast}(p, q) \wedge \alpha(p) = f \wedge \alpha(q) = \varnothing \rightarrow \alpha(p) = f$.

Let us consider a frequent case of casting an object in the fragment of code shown below.

```
int index;
SomeClass o;
......................
vectorObj.put( o );
......................
ParentClass pc = (ParentClass)
```

The `ParentClass` class is the parent of the class SomeClass which is mapped to the set of features `F`. The class `ParentClass` is mapped to a different set of features `G`. According to the basic rule, the object `o` is annotated with the feature label `F`. The object `o` is stored in a vector represented by the object `vectorObj`. When retrieved from the vector the object `o` is cast to the class `ParentClass`, and its annotation changes to the feature label `G`.

Sometimes a default mapping should be overwritten. For example, a class may be mapped to one feature, but its instances should be mapped to some other features. The following is the rule to handle this condition.

Containment: if an object p is a member of type q that is annotated with feature label f, then the object p is also annotated with the feature label f: $Contain(p, q) \wedge f \in \alpha(q) \rightarrow f \in \alpha(p)$.

Instance overriding: annotation of an object overrides the default feature labels assigned to this object by the basic rule: $Type(o) = \tau \wedge f \in \alpha(\tau) \wedge g \in \alpha(o) \rightarrow f \in \alpha(\tau) \wedge g \in \alpha(o)$.

Member overriding: the mappings for members of the containment types can be overwritten: $Contain(p, q) \wedge f \in \alpha(p) \wedge g \in \alpha(q) \rightarrow f \in \alpha(p) \wedge g \in \alpha(q)$.

Precedence: if the containment type is mapped to one feature and the type of a member variable of this containment is mapped to a different feature, then this variable is mapped to the same feature to which its type is mapped: $Contain(p, o) \wedge f \in \alpha(p) \wedge g \in \alpha(q) \wedge Type(o) = q \rightarrow g \in \alpha(o)$.

Interface: when programmers map interfaces to features, these mappings are preserved in classes that implement mapped interfaces: $Subtype(p, q) \wedge f \in \alpha(q) \wedge Contains(q, z) \rightarrow f \in \alpha(z) \wedge Contains(p, z)$. That is, if fields or methods are declared in an interface that is mapped to some features and is implemented by some class that is mapped to different features, then the interface fields and methods inherit the interface feature mapping.

Inheritance: if a class extends some other class that is mapped to some feature, then the extended class is automatically mapped to the same feature: $Inherit(p, q) \wedge f \in \alpha(q) \wedge \alpha(p) = \emptyset \rightarrow f \in \alpha(p)$. This rule is dictated by the Java idiom of inheritance stating that a class may implement may interfaces, but it can extend only one class. The extended class can be explicitly remapped to a different feature without affecting the mapping defined for the parent class.

3.3 The Formal Problem Statement

When programmers specify the initial mapping between program entities and features, they define a partial function α_0 over the domain of program entities $\pi_0 \subseteq \pi$ and a range of features $f_0 \subseteq f$. The goal of Lean is to compute the partial function α_1 over the subset of the domain of program entities $\pi_1 \subseteq \pi$ and the range of features $f_1 \subseteq f$ abiding by the rules specified in Section 3.2. Rules of Program Annotations, such that $\pi_0 \subseteq \pi_1$ and $f_0 \subseteq f_1$ and $\forall$ $(a, b) \in \delta$, $a,b \in \pi$, $c,d \in f$, s.t. $c \in \alpha(a) \wedge d \in \alpha(b)$ $\mid \gamma \in (c, d)$. That is, if program entities a and b are annotated with feature labels c and d respectively, and these entities are related to each other via some δ-relation, then the features labeled c and d in some FD should be related via some γ-relation.

4. Lean Architecture

The architecture for Lean and a brief process description are shown in Figure 3. Solid arrows show the process of annotating program entities with feature labels, and dashed arrows specify the process when training the Learner. The inputs to the system are program source code and a Feature Diagram (FD). The Mapper is a *Graphic User Interface (GUI)* tool whose main components are Java and XML parsers, program and FD analysis routines, a rule-based engine, and an instrumenter. A Java parser produces a tree representing program entities. Since FDs are represented as XML data, the Mapper uses an XML parser to produce a tree representing features and variation points in the FD. The Mapper GUI presents both the FD and the source code using tree widgets. Programmers use the Mapper GUI to specify initial mappings between features from the FD and program entities from the source code.

Once the Mapper presents the FD and program parse trees, the user specifies initial mappings between features and program entities by establishing links between program entities from the program tree with features. From these mappings the Mapper generates the initial annotations. The user can also specify the entities that should not be annotated and therefore excluded from the annotation process. For example, using Lean to annotate an integer variable counting the number of iterations in a loop consumes computing resources while there may not be an appropriate feature for annotating this variable, or annotating it does not warrant the amount of work required. Variables whose values contain binary (nonprintable) characters should also be excluded from monitoring since machine learning algorithms are not yet effective for classifying these variables.

These initial annotations are expanded using the rules from Section 3.2. Rules of Program Annotations in two steps. First, relations defined in Section 3.1. Definitions are built using the Mapper's program analysis routines. Then the Mapper's rule engine analyzes these relations and initial annotation links, and expands these initial annotations to other program entities. For example, if a class is mapped to some feature, then, according to the containment rule, its members are

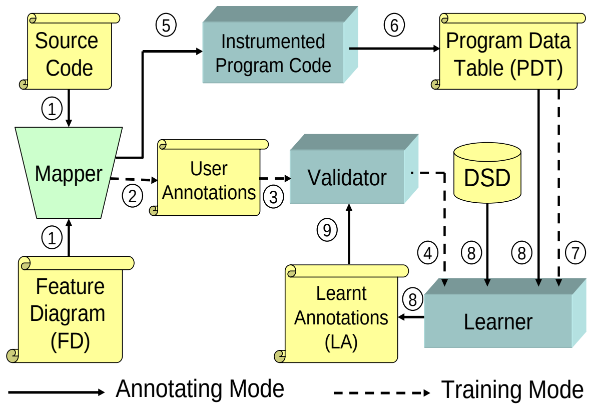


Figure 3. Lean Architecture and its process.

1) Programmers annotate the Source Code with concepts from the FD using the Mapper
2) The Mapper produces initial mappings
3) The Validator validates initial mappings
4) The Validator supplies the initial annotations to the Learner for training
5) The Mapper instruments and compiles the source code
6) The program runs and its instrumentation outputs the Program Data Table (PDT)
7) Annotated variables and their values from the PDT are supplied to the Learner for training
8) Learner classifies program variables from the PDT and produces learnt annotations (LA) with the help of Domain-Specific Dictionary (DSD)
9) The Validator validates LAs and uses negative examples to retrain the Learner

mapped to the same feature by default, and according to the basic rule, all instances of this class are also annotated with this feature label.

The Mapper outputs user annotations in an XML file that consists of a table whose entries are program entities and their annotations, and δ- and γ-relations. The latter is obtained from the source code and the FD using the Mapper's program and FD analysis routines. This XML file is passed to the Validator. Recall the direct mapping rule stating that for a δ-relation between annotated objects in the source code there is a γ-relation between the corresponding features. The Validator validates the initial mappings by matching these relations when both components of δ-relations are annotated. When the initial annotations are validated, the Validator supplies them to the Learner for training. Lean uses variable names and the corresponding feature labels to train the Learner to classify program entities by their naming patterns.

The Mapper instruments the source code to record run-time values of unannotated variables, and calls a Java compiler to produce an executable program. Then the program runs storing names and the values of program variables in the *Program Data Table (PDT)*. This training uses the content of the annotated variables rather than their names. Once the Learner is trained, it classifies unannotated program variables. These variables are supplied to the Learner as the columns of the PDT. In addition, *Domain-Specific Dictionaries (DSDs)* increase the precision of the classification. The output of the Learner is a set of *learnt annotations (LAs)*. Some of these annotations may be incorrect because the Learner does not guarantee 100% precision. Finding incorrectly learnt mappings in a large program is a tedious and a laborious exercise. The Validator automates this process by taking in the learnt program annotations and validating these annotations by exploring relations between variables in the source code and features in the FD. The output of the Validator is a list of rejected annotations which the Learner may use to improve its predictive capabilities.

5. Learning Annotations

This section shows how Lean learns and validates annotations using run-time monitoring, program analysis, and machine learning. We describe the key idea, present the organization of the learner, and give the learning algorithm. This section ends with a discussion on how to extend the learner to adapt to other domains.

5.1. Our Approach

In our approach, automating the program annotation process is treated as a classification problem: given `n` features and a program variable, which feature matches this variable the most? Statistical measures of matching between variables and features are probabilistic. The Learner classifies program entities with the probabilities that certain feature labels can be assigned to them. By taking a set of annotated program variables and their values, a classifier is built and trained to classify an unannotated variable based on the information learned from the annotated variables.

Initially, all unannotated variables have equal and arbitrary chosen probabilities (in the interval from 0 to 1) of matching given feature labels. For example, given three features Email, Web, and Phone the variable `S[3]` is assigned three initial probabilities $p_{Email}(S[3]) = p_{Web}(S[3]) = p_{Phone}(S[3]) = 1/3$. Assigning equal probabilities reflects our lack of knowledge of what feature should match a specific variable. When classifying this variable these probabilities are recomputed. If the name of the variable matches the name of the feature, then the probability of annotating this variable with the feature increases. If the values of `S[3]` do not contain any digits, then the probability that this variable is annotated with the `Phone` concept may decrease to zero. After classifying the program variable `S[3]`, a learner may assign the probability $p_{Email}(S[3]) = 0.7$ that the variable `S[3]` represents the `Email` concept, and probabilities $p_{Web}(S[3]) = 0.3$ and $p_{Phone}(S[3]) = 0.1$. Since the probability that the variable `S[3]` represents the `Email` concept is the highest, the learner will issue a prediction that this variable should be annotated with the `Email` concept.

5.2. The Organization of the Learner

Lean has its roots in the *Learning Source Descriptions (LSD)* system developed at the University of Washington [6] for reconciling schemas of disparate XML data sources. The purpose of LSD is to learn one-to-one correspondences between elements in XML schemas. Lean employs the LSD multistrategy learning approach [7, 8], which organizes multiple learners in layers. The learners located at the bottom layer are called base learners, and their predictions are combined by metalearners located at the upper layers. We use three types of learners: name, content, and Naïve Bayes. Even though many different types of learners can be used with the multistrategy learning approach, we limit our study to these three types of learners since they proved to give good results when used in LSD.

We illustrate the multistrategy learning approach with the following example. One base learner $\mathtt{BL}_1$ may issue a prediction that the variable `Pho` from the example shown in Figure 1 matches feature `Address` with the probability `0.3`, feature `Email` with the probability `0.1`, and the feature `Phone` with the probability `0.7`. We write these matches as variant $\langle \mathtt{Address{:}0.3,\ Email{:}0.1,\ Phone{:}0.7} \rangle^{\mathtt{BL}_1}_{\mathtt{Pho}}$, where the field labels are feature labels and field values are the probabilities of matching a given variable that is specified as a subscript to the variant. The superscript of the invariant shows the name of the learner used to classify a given variable. The other base learner $\mathtt{BL}_2$ may issue a different prediction $\langle \mathtt{Address{:}0.2,\ Email{:}0.3,\ Phone{:}0.9} \rangle^{\mathtt{BL}_2}_{\mathtt{Pho}}$. A metalearner combines these predictions by multiplying the probabilities by weights assigned to each learner and taking the average for the products for the corresponding labels of the predictions for the same program variable. Thus, the resulting prediction issued by a metalearner in our example is $\langle \mathtt{Address{:}0.25,\ Email{:}0.2,\ Phone{:}0.8} \rangle^{\mathtt{ML}}_{\mathtt{Pho}}$ with weight equal to 1 for both learners. Based on this prediction, the feature label `Phone` matches the program variable

Pho with the highest probability 0.8, and based on this result the metalearner assigns the annotation Email to this variable.

There are three types of base learners used in the Lean learner: a name matcher, a content matcher, and a Bayes learner. Here we give a brief description; these learners are described in detail in [6, 10]. Name matchers match the names of program entities with feature labels. The name matching is based on *Whirl*, a text classification algorithm based on the nearest-neighbor class [9]. This algorithm computes the similarity distance between the name of a program entity and a feature label. This distance should be within some threshold for the name. This threshold value is determined when the learner is trained on selected data.

Whirl-based name matchers work well for meaningful names especially if large parts of them coincide or they are synonyms. They do not perform well when names are meaningless or consist of combinations of numbers, digits, and some special characters (e.g., underscore or caret). For example, Whirl is unable to correctly classify the variable S[3] shown in Figure 1c.

Content matchers work on the same principles and use the same algorithm (Whirl) as name matchers. The difference is that content matchers operate on the values of variables rather than their names. Content matchers work especially well on string variables that contain long textual elements, and they perform poorly on binary (nonprintable) and numeric types of variables. For example, values of S[3] may contain sentence “Email to: John@ax.com”, and the presence of the word Email indicates that this variable should be classified as the Email concept.

Finally, *Bayes* learners, particularly the *Naïve Bayes classifier*, are among the most practical and competitive approaches to classification problems [10]. Naïve Bayes classifiers are studied extensively [10, 11], so we only state what they do in the context of the problem that we are solving here. For each variable var_j we parse its values into words and create a bag of words that the values of this variable take. Given feature labels $\{f_1, \ldots, f_m\}$ the Naïve Bayes classifier assigns var_j to some feature label f_k, $1 \leq k \leq m$, such that the probability that $p(f_k \mid var_j)$ that the variable var_j belongs to the feature f_k, is maximized.

5.3. Overview of Learning Algorithm

Lean learning algorithm consists of two phases: the training phase and the annotating phase. The training phase improves the ability of the learners to predict correct annotations for program variables. Trained learners classify program variables with feature labels, and based on these classifications, Lean annotates programs. The accuracy of the classification process depends upon successful training of the Learner.

To disambiguate variables that are given the same names in different scopes (i.e., program text regions in which variables bindings are active), each variable is identified with its access path. For example, if a variable named `var` is declared in the method `M` of the class `C` which is defined in the package `P`, then the access path to this variable is `P.C.M.var`.

The data for training learners come from an instrumented program. When it runs, the instrumented code outputs variable names and their values into the program data table (PDT). Recall that this table contains columns for access paths, and the cells for these columns are filled with values that the access path destination variables take during program runs. During the training phase, weights of the base learners are adjusted and probabilities are computed for each learner using the PDT columns containing data for annotated variables. Then, during the classification step the previously computed weights and probabilities are used to predict the feature label for unannotated variables.

Each base leaner is assigned a weight (a real number between `0` and `1`) that characterizes its accuracy in predicting annotations of program variables. Initially, all weights are the same. For each classified program entity the weights of the learner are modified to minimize the squared error between the predefined value (i.e., `1` if the prediction is correct, or `0` otherwise) and the computed probability multipled by the weight assigned to the learner. This approach is called regression [10].

Most machine learning approaches are as good as the selected training data. Selecting test data is often characterized by how well the data represents the true value distribution. Overfitting training data is one of the most common mistakes. Examples of overfitting is overestimating the importance of some rare words or selecting a set of training data that covers test data. If the latter occurs, then the classifier cannot be correctly evaluated since it is important to test its predictive capabilities on data that was not used for training.

Lean uses the cross-validating [10] training data it divides into few pairs of training and testing sets. Then, each learner is trained for each pair of training and testing data sets, and the results are averaged to produce a more accurate estimate.

5.4. Learning Conditional Annotations

Consider a fragment of code shown in Figure 4. The `while` loop iterates over the integer variable `counter` whose value modulo two serves as an input to the method `GetAttribute`. This method interates through some dataset and returns `String` type values which are assigned to the variable `var`. Suppose that the value returned by the method `GetAttribute` belongs to the concept `Address` when the value of the variable `counter` is even and to the concept `Email` when the value is odd. It means that the variable `var` should be annotated with these two concepts. However, these annotations are conditional upon the value of the counter.

Temporary variables that are incremented by a predictable amount each time through the loop, called *induction variables* [11]. Examples are variables whose definitions within the loop are of the form `counter = counter + c`, where `c` is a loop invariant. Lean combines induction variable values with the values of other variables to train the Learner, and subsequently classify the unannotated variables that depend on these induction variables. For the code fragment shown in Figure 4 values for the variable `var` are used in conjunction with the values of the expression `counter%2` when training the Learner and later classifying the variable `unannotated`.

```
int counter = 0;
String var, unannotated;

while(  counter++ < SomeNumber ) {
        var = GetAttribute( counter % 2 );
        unannotated = GetAttribute( (counter+1) % 2 );
}
```

Figure 4. Example of code requiring conditional annotations.

5.5. Extending the Learner

Domains use special terminologies whose dictionary words mean specific things. Programmers use domain dictionaries to name variables and types in programs written for these domains. For example, when word "wafer" is encountered in a value of some variable of a program written for a semiconductor domain, this variable may be annotated with the wafer concept. Many domains have dictionaries listing special words, their synonyms, and explaining their meanings. In addition, these domain-specific dictionaries may specify constraints that can be used to improve the precision of program annotations. For example, a list of diameters of wafers permitted by certain standards can be included in the definition of the wafer in a semiconductor domain dictionary.

Lean incorporates the knowledge supplied by these dictionaries. Each concept in these dictionaries has a number of words that are characteristic of this concept. If a word from the dictionary is encountered in a value of a variable, then this variable may be classified and subsequently annotated by this concept. We use a simple heuristic to change the probabilities that variables should be annotated with certain feature labels. If no dictionary word is encountered among the values of a variable, then its probabilities remain unchanged. Otherwise, if a word belongs to some concept, then the probability that the given variable belongs to this concept is incremented by some small real number Δ_p, i.e., $p_{concept}$`(var) = `$p_{concept}$`(var) + `Δ_p. We choose this number experimentally as `1/(# of words in a DSD)`. If the resulting probability is greater than `1.0` after adding Δ_p, then the probability remains `1.0`.

6. Inferring and Validating Annotations

This section describes how we infer and validate program annotations using program analysis. First, we state the rationale for inferring and validating annotations. Then, we illustrate the core idea for inferring and validating annotation on an illustrative example. Next, we give the algorithm for inferring annotations for δ-relations containing annotated and unannotated components. Finally, we show how to validate program annotations.

6.1. The Rationale

Lean cannot annotate all variables due to a number of factors. Machine learning approaches are only as good as the training data, and they do not guarantee 100% classification accuracy. Some variables cannot be classified because they take hard-to-analyze values. Examples are variables whose values are binary (nonprintable) strings, or integer variables holding values for salaries and zip codes. The former makes it difficult to train classifiers since patterns in binary data are inherently complex. The latter example demonstrates that when values of two variables are approximately the same, it is difficult to train the classifier to recognize these variables by their values.

Algorithms for inferring and validating annotations predict annotations for partially annotated δ-relations (i.e., when one component of a δ-relation is annotated, and the other is not) and detect when incorrect annotations are assigned in certain situations. These algorithms are not sound. They can miss incorrect annotations and they cannot pinpoint the source of the fault that led to incorrect annotations. However, these algorithms perform well in practice for the majority of cases as our experiments show.

6.2. An Illustrative Example

Consider an expression `c = d + e`, where `c`, `d`, and `e` are variables, and a mandatory FD whose features `p` and `q` are attached to the base feature `f`. Since variables `c`, `d`, and `e` are related via the δ-relation, we write `δ(c, d)`, `δ(c, e)`, and `δ(d, e)`. We write `γ(f, p)` and `γ(f, q)` for features `f`, `p`, and `q` since they are related. The variable `c` is mapped to the feature `f` during the initial mapping, i.e. `f` ∈ α(`c`).

Consider some other expression `a = b + d`, where `a`, `b`, and `d` are variables, and a mandatory FD whose features `v` and `p` are attached to some other feature `u`. This expression is defined in the same scope as the previous expression, and the variable `d` is the same in both expressions. Feature `p` is shared by both FDs. Since variables `a`, `b`, and `d` are related via the δ-relation, we write `δ(a, b)`, `δ(a, d)`, `δ(b, d)`. We write `γ(u, p)` and `γ(u, v)` for features `u`, `p`, and `v` since they are related too. The variable `a` is mapped to the feature `u` during the initial mapping, i.e. `u` ∈ α(`a`).

Our goal is to validate the annotation assigned by the Lean classifiers to the variables in the given expressions. The solution to this validation problem is to infer sets of possible annotations, and then determine whether assigned annotations exist in the inferred sets. If they exist, then Lean assigned annotations correctly.

Let us illustrate the solution to the validation problem. Let us apply the annotation function to the δ-relations δ(c, d) and δ(c, e): α(δ(c, d)) = γ(α(c), α(d)) and α(δ(c, e)) = γ(α(c), α(e)). By substituting the initial mapping α(c) = f, we obtain γ(f, α(d)) and γ(f, α(e)). Recall from the main rule stated in Section 3.2. Rules of Program Annotations that for each δ-relation between objects in the source code which are annotated with feature labels there is a corresponding γ-relation between the corresponding features in some feature diagram. We make the annotation function α total by adding concept ? to its range. All unannotated program entities are mapped by α to the concept ?.

Since no annotation is defined for variables d and e, we replace α(d) and α(e) with ? in relations γ(f, α(d)) and γ(f, α(e)) obtaining the pattern relation γ(f,?). In this pattern relation the symbol ? is used to match the second unknown component. To find features that are the values of the unknown component, we should find γ relations whose first component is the feature label f. Then, the second components of the δ-relations can be mapped to the second components of these γ relations. Specifically, α(d) = α(e) = {{p},{q}}. Repeating the same process for the other expression we obtain α(b) = α(d) = {{p},{v}}. Thus, possible annotations for variables b, d, and e have been inferred. We can make the set of possible annotations more precise for the variable d. Since both expressions are located in the same scope, the variable d should be mapped to the same sets of features. By taking the intersection of the sets to which the variable d is mapped, we obtain α(d) = {p}. If the intersection yields an empty set, then the annotations of the variable d are flagged as possible errors.

Suppose that after running Lean it annotates the variable e with the feature label v. In order to validate whether this annotation is correct, we evaluate if the feature v exists in the set of possible annotations for the variable e.

6.3. The Annotation Inference Algorithm

The algorithm InferAnnotations for inferring annotations is shown in Figure 5. Its input is the set of δ-relations, γ-relations, and mappings α between program entities and sets of features. The algorithm iterates through all δ-relations to find partially annotated ones (i.e., when one component of a δ-relation is annotated, and the other is not). Then, for each found δ-relation the annotation function is applied to the annotated component to obtain the set of features with which this component is annotated. Then, for each feature in this set all γ-relations are found whose component matches this feature. The other components of the obtained γ-relations make it into the set of possible annotations with which the unannotated component of the δ-relation may be annotated.

The main `for`-loop of the algorithm explores all δ-relations. It checks each variable in each δ-relation to see if it is annotated. If the annotating set of features is empty for one

```
InferAnnotations( $\delta, \gamma, \alpha$ )
for each $(a, b) \in \delta$ do
    $\alpha(a) \mapsto f_k$
    $\alpha(b) \mapsto f_m$
    if $f_k = \emptyset$ then
        for each $(c, d) \in \gamma$ do
            if $d \in f_m$ then
                $f_k \mapsto f_k \cup \{c\}$
            else if $c \in f_m$
                $f_k \mapsto f_k \cup \{d\}$
            endif
        endfor
    else if $f_m = \emptyset$ then
        for each $(c, d) \in \gamma$ do
            if $d \in f_k$ then
                $f_m \mapsto f_m \cup \{c\}$
            else if $c \in f_k$
                $f_m \mapsto f_m \cup \{d\}$
            endif
        endfor
    endif
    if $\exists (u,v) \in \delta$ s.t. $u = a \vee v = a$ then
        $\alpha(u) \mapsto f_s$
        $\alpha(u) = f_s \cap f_k$
        $\alpha(a) = f_s \cap f_k$
    endif
endfor
```

Figure 5. Algorithm for inferring additional annotations.

variable and nonempty for the other, then all γ-relations are searched whose component is a subset in the annotating set of features. If such γ-relation is found, then its other component is added to the annotation set of the unannotated variable in the δ-relation.

The last `if` condition in the algorithm deals with the same variable, namely `a`, used in two or more expressions in the same scope. This variable may be annotated differently. In this case an intersection is taken of the feature sets with which the uses of this variable are annotated. The result of this operation is an empty feature set, a reduced feature set, or the full set if the annotations coincide.

6.4. The Validating Algorithm

The `ValidateAnnotation` algorithm shown in Figure 6 validates whether annotations are correct. The key criteria for validating annotations is to check that for each δ-relation between annotated entities in the source code there is a corresponding γ-relation between the corresponding features in some feature diagram. The input to this algorithm is the set of δ-relations, γ-relations, and mappings α between program entities and sets of features. Each δ-relation has a color associated with it which is initially set to red. The red color means that a given δ-relation is not correctly annotated, and the green color means that all components of a given δ-relation are annotated correctly, or not all components are annotated.

The outer `for`-loop iterates through all δ-relation, and annotations f_k and f_m for components of each relation are obtained. If one or both components of a given δ-relation are not annotated, then this relation is colored green. Otherwise, the inner `for`-loop searches through γ-relations to find one whose components are members of f_k or f_m annotation sets. If such a γ-relation exists, then the corresponding δ-relation is colored green and the inner loop is exited. Otherwise, if the inner loop exits without finding a γ-relation whose components are members of f_k or f_m annotation sets, then the δ-relation is red and may not be valid.

This algorithm does not specify what the correct annotation of a program variable is or what caused the error in program annotation. In fact, annotation errors many be caused by incorrect feature diagram, errors in program source code, or both. The last `for`-loop iterates through all δ-relations, checks the colors, and prints red relations as potentially incorrect.

```
ValidateAnnotations( $\delta, \gamma, \alpha$ )
for each $(a, b) \in \delta$ do
    $color((a, b)) \mapsto red$
    $\alpha(a) \mapsto f_k$
    $\alpha(b) \mapsto f_m$
    if $f_k \neq \emptyset \wedge f_m \neq \emptyset$ then
        for each $(c, d) \in \gamma$ do
            if $(c \in f_m \wedge d \in f_k) \vee (c \in f_k \wedge d \in f_m)$ then
                $color((a, b)) \mapsto green$
                break;
            endif
        endfor
    else
        $color((a, b)) \mapsto green$
    endif
endfor
for each $(a, b) \in \delta$ do
    if $color((a,b)) \neq green$ then
        print error
    endif
endfor
```

Figure 6. Algorithm for validating annotations.

6.5. The Computational Complexity

Suppose a program has n variables and a feature diagram has m features. Then it is possible to build $n(n-1)/2$ δ-relations and $m(m-1)/2$ γ-relations. Thus, the space complexity is $O(n^2 + m^2)$. The time complexity is deterimed by two `for`-loops in the `ValidateAnnotation` and `InferAnnotations` algorithms. The external `for`-loops iterate over all δ-relations and the internal `for`-loops iterate over all γ-relations. Considering all other operations in the algorithms taking $O(1)$, the time complexity is $O(n^2m^2)$.

7. The Prototype Implementation

The prototype implementation of Lean is based on its architecture shown in Figure 3. Its main elements are the Mapper, the Learner, and the Validator. The Mapper is a GUI tool written in C++ that presents Java source code in a tree format along with feature diagrams which are represented in XML format. The Mapper includes an EDG Java parser [13] and an MS XML parser. An example of FD in XML format is shown in

Figure 7. The root of the XML data is the tag `FD` whose attribute `Name` specifies the name of the application to which this FD is applicable. The child element of the root is `Feature` with the attribute `Name` whose value if the name of the feature. Children tags `VarPoint` describe variation points to which other features are attached. Each element `VarPoint` has the attribute `Type` whose value specifies the type of a feature attachment, i.e., mandatory, optional, alternative, or or-feature. If the type of the variation point is alternative or or-feature, then this `VarPoint` element contains children `VarPoint` elements whose types are mandatory or optional.

```
<FD Name=”VMT”>
        <Feature Name=”Vendor”>
                <VarPoint Type=”Alternative”>
                        <VarPoint Type=”Mandatory”>
                                <Feature Name=”Phone”/>
                        </VarPoint>
                        <VarPoint Type=”Optional”>
                                <Feature Name=”Email”/>
                        </VarPoint>
                </VarPoint>
        </Feature>
</FD>
```

Figure 7. Representation of a feature diagram in the XML format.

Programmers map features to program entities using the Mapper GUI which outputs an XML file whose example content is shown in Figure 8. The root of the XML data is the element `Annotations` whose attributes `FD` and

```
<Annotations FD=”VMT” Program=”vendors.java”>
        <Annotation Entity=”Type”>
                <AccessPath Type=”Class”>vendors</AccessPath>
                <Feature>Vendor</Feature>
        </Annotation>
        <Annotation Entity=”Field”>
                <AccessPath Type=”String”>vendors.Pho</AccessPath>
                <Feature>Phone</Feature>
        </Annotation>
        <Exclusions>
                <AccessPath Type=”String”>vendors.Email</AccessPath>
        </Exclusions>
</Annotations>
```

Figure 8. XML file containing annotations of program entities.

Program specify the feature diagram used and the name of the programs whose variables are annotated, respectively. Each variable is specified by its access path, and features are specified by their names. The Exclusions element contains a list of variables that should be excluded from the annotation process. These variables are described by children elements AccessPath of the element Exclusions.

The Mapper obtains δ and γ relations from the source code by applying control and data flow analyses. Obtaining γ-relations from FDs is much simpler than δ-relations since checking only the parent and children of VarPoint elements is required. The output of these analyses is an XML file containing γ and δ relations as shown in Figure 9. The root element Relations has children elements Deltas and Gammas. These elements contain the collections of δ and γ relations respectively. The Delta elements are used to specify δ-relations and the Gamma elements specify γ-relations.

The Mapper instruments the source code by adding calls that log runtime values of program variables. These are the variables that are not annotated initially, and whose annotations should be learned from the content of their values. Runtime logging is added after the definitions of the variables and after statements and expressions to where the monitored variables are assigned. Lean's data flow analysis framework locates variable definitions and traces the uses of these variables until either the end of the scope for the definitions or the definition of new variables overwrite previous definitions. Only distinct values of the monitored variables are collected.

Once the initial mapping is complete, the Mapper sends the XML annotation and relation file rules to the Validator. Algorithms InferAnnotations and ValidateAnnotations constitute the core of the Validator. It uses the algorithm InferAnnotations to add possible annotations to program entities, and then it applies the ValidateAnnotations algorithm to validate annotations assigned by programmers or by the Learner.

Both the Validator and the runtime logger code output data in the *Attribute Relation File Format (ARFF)* file format. ARFF serves as an input to the Learner which is based on a machine-learning Java-based open source system WEKA [14][15]. The Learner is trained on the collection of data from program runs. The collection is done by instrumenting program source code and running the program with statements that log the run-time

```
<Relations FD=”VMT” Program=”vendors.java”>
        <Deltas>
                <Delta ID=”1”>
                        <AccessPath Type=”String”>vendors.Pho</AccessPath>
                        <AccessPath Type=”String”>vendors.Email</AccessPath>
                </Delta>
        </Deltas>
        <Gammas>
                <Gamma ID=”2”>
                        <Feature>Vendor</Feature>
                        <Feature>Phone</Feature>
                </Gamma>
        </Gammas>
</Relations>
```

Figure 9. XML file containing δ and γ relations.

values of program variables in the ARFF file called Program Data Table in the Lean architecture.

Recall the code fragment shown in Figure 1a which contains the definition of the class `vendors`. A user provides the initial annotation `Address` for the variable `Add`. After the instrumented program is run, the run-time logging code outputs training data in the ARFF format whose sample is shown in Figure 10. This ARFF file contains the values taken by the variable `Add` at the runtime. The structure of ARFF files is described in detail in [14][15].

```
%ARFF file for training the Lean Learner
@relation AddressTraining
@attribute Add string
@attribute concept? {Address, Unknown}

@data
“Tamara Circle, Austin”, Address
“McNeil Drive, Austin”, Address
“xxx 123 yy”, Unknown
“Sims Road, Dallas”, Address
```

Figure 10. A sample ARFF fileFigure 4.

The ARFF file can be viewed as a table with attributes. Its header contains the keyword `@relation` followed by the name of the training set `AddressTraining`, and a series of attributes prepended with the keyword `@attribute`. Two attributes are shown in Figure 10: the attribute `Add` followed by its type `string`, and the nominal attribute `concept` followed with the `?` sign meaning that it is used to classify the attribute `Add`. Two nominal values `Address` and `Unknown` for the attribute `concept` give us two choices of classifying program variables. The data instances of these attributes follow the section definition `@data`. This data instances in this section are produced as a result of run-time monitoring and manual classifying each data instance.

The classification of unannotated variables is accomplished by obtaining runtime values of these variables and supplying them to the Learner which emits predictions for feature labels with which these variables should be annotated. These predictions are refined during the validation stage by manual inspection. The refined predictions are supplied to the Learner for training to improve its accuracy. This continuing process of annotating, validating annotations, and learning from the validated annotations makes Lean effective for long-term evolution and maintenance of software systems.

8. Experimental Evaluation

In this section we describe the methodology and provide the results of experimental evaluation of Lean on open-source Java programs.

8.1. Subject Programs

We experiment with a total of seven Java programs that belong to different domains. MegaMek is a networked Java clone of BattleTech, a turn-based sci-fi boardgame for two or more players. PMD is a Java source code analyzer which finds unused variables, empty catch blocks, etc. FreeCol is an open version of a Civilization game in which the player conquers new worlds. Jetty is an Open Source HTTP Server and Servlet container. The Vehicle Maintenance Tracker (VMT) tracks the maintenance of vehicles (e.g., boats, cars, and planes). The Animal Shelter Manager (AMS) is an application for animal sanctuaries and shelters that includes document generation, full reporting, charts, internet publishing, pet search engine, and web interface. Finally, Integrated Hospital Information System (IHIS) is a program for maintaining health information records.

8.2. Methodology

To evaluate Lean, we carry out two experiments to explore how effectivly Lean annotates programs and how training affects the accuracy of predicting annotations. We also investigate the cases the validation algorithm rejected annotations.

In the first experiment, we create a domain-specific dictionary (DSD) and a feature diagram (FD) for each subject program. Then we annotate a subset of variables for each program and run Lean to annotate the rest of the program. The goal of this experiment is to determine how effective Lean is in annotating program variables for programs of different sizes and from different domains. Each annotation experiment is run with and without a DSD in order to study the effect of the presence of DSDs on the quality of Lean annotations.

We measure the number of variables annotated by Lean as well as the number of annotations rejected by the validating algorithm. The number of variables that Lean can possibly annotate, `vars`, is `vars = total – (excluded + initial)`, where `total` is the total number of variables in a program, `excluded` is the number of variables excluded from the annotation process by the user, and `initial` is the number of variables annotated by the user. Lean's accuracy ratio is computed as `accuracy = (vars – rejected)/vars`, where `vars` is the number of variables annotated by Lean and `rejected` is the number of annotations rejected by the validating algorithm.

The goal of the second experiment is to evaluate the effect of training on the Lean's classification accuracy. Specifically, it is important to see the amount of training involved to increase the accuracy of annotating programs. Training the Lean Learner is accomplished by running instrumented programs with distinct sets of input data. If the Learner should be trained continuously, then certain applications may be exempt from our approach. On the contrary, if a program should run a reasonable number of times in order to collect distinct data sets for training and classification, then our approach is practical and can be used in the industrial settings.

8.3. Results

Table 2 contains the name of a program, the size of the DSD, the number of lines of code, the number of features in an FD, the number of variables that Lean could potentially annotate, the percentage of initial annotations computed as ratio `initial/total`, where `total` is the total number of variables in a program, and `initial` is the number of variables annotated by users. The next two columns compare the percentage of total annotations without and with the DSD. The last column of Lean this table shows the accuracy of Lean when used with DSDs.

The highest accuracy is achieved with programs that access and manipulate domain-specific data rather than general information without a strong influence of any domain terminology. The lowest level of accuracy was with the program PMD which analyzes Java programs and it not based on any specific domain. The highest level of accuracy was achieved with the programs ASM and VMT which are written for specific domains with well-defined terminologies, and whose variable names are easy to interpret and classify.

PROGRAM	SIZE OF DSD, WORDS	LINES OF CODE, LOC	# OF FEAT-URES	NUM-BER OF VARS	USER ANNOTA-TIONS, %	LEAN ANNOTS W/O DSD	LEAN ANNOTS WITH DSD	ACCU-RACY, %
Megamek	60	23,782	25	328	10%	58%	64%	64%
PMD	20	3,419	12	176	7.4%	23%	34%	35%
FreeCol	30	6,855	17	527	4.7%	56%	73%	79%
Jetty	30	4,613	6	96	12.5%	42%	81%	52%
VMT	80	2,926	8	143	5.6%	65%	72%	83%
ASM	60	12,294	23	218	5.5%	57%	79%	87%
IHIS	80	1,883	14	225	8%	53%	66%	68%

Table 2. Results of the experimental evaluation of Lean on open source program.

The next experiment evaluates the accuracy of the Lean learner. Figure 11 shows that when annotating the AMS application, the Learner achieved the highest accuracy, close to 90%. This accuracy was achieved when the number of distinct training samples reached 500. The results of this experiment show that applications need to be run only few hundred times with different input data in order to train the Learner to achieve good accuracy. Since most applications run at least several thousand times during their testing, using Lean as a part of application testing to annotate program source code is practical. Potentially, if a low-cost mechanism is applied to collect training samples over the life time of an application, then Lean can maintain and evolve program annotations with the evolution of programs.

Finally, we used the Learner trained for the VMT application to annotate variables in other applications. This methodology is called *true-advice* versus *self-advice* which uses the same program for training and evaluation. Figure 12 shows the percentage of variables that the Lean Learner annotates with self-advise (left bar) versus the true-advice annotations (right bar) when the Learner is trained on the VMT application. This experiment shows that Lean can be trained on one application and used to annotate other programs if they operate on the same domain-specific concepts. ASM and IHIS have common concepts with the VMT application, and it allows learners to be trained and used interchangeably achieving the high degree of automation in annotating program variables.

9. Related Work

The importance of annotations for program checking and verification is emphasized in [16]. In addition to the benefits outlines in Section 1, getting annotations into programs is important for developing algorithms and techniques for the verifying compiler. Although many notable publications are written on the use of annotations for various purposes, only few of them describe approaches and tools for automating the program annotation process.

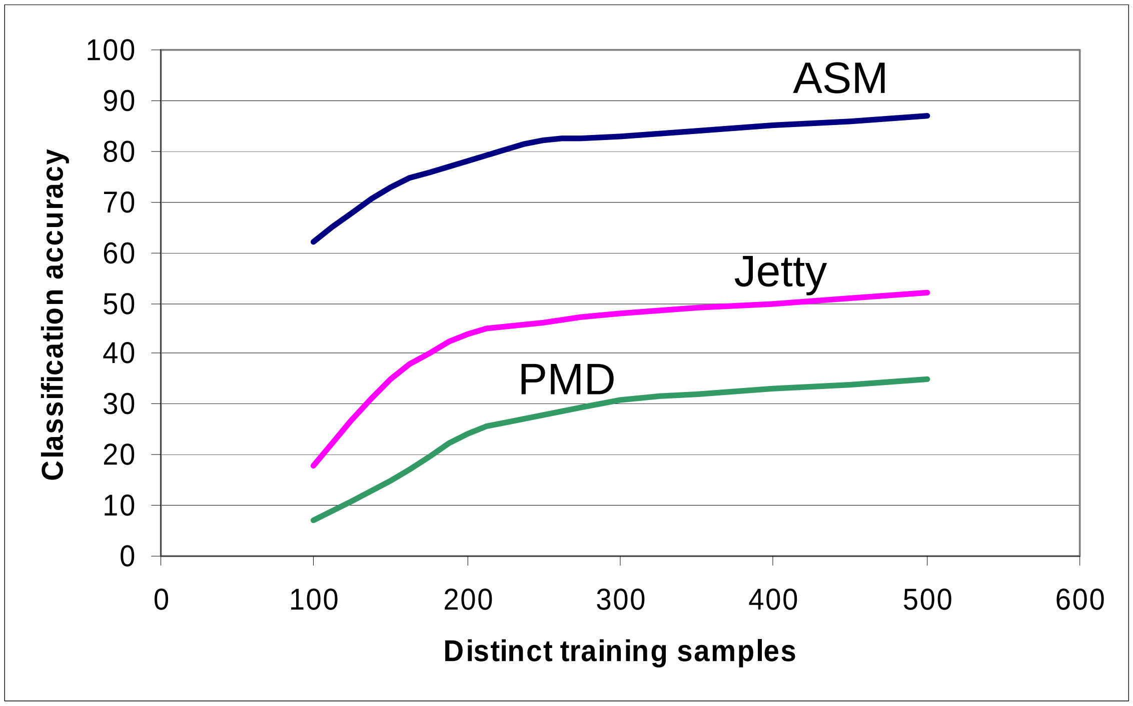


Figure 11. The graph of the accuracy of the Lean learner.

One of the earliest papers on automating program annotations [17] describes a technique for annotating Algol-like programs automatically given their specifications. The annotation techniques are based on the Hoare-like inference rules which derive invariants from the assignment statements, from the control structure of the program, or, from suggested invariants. Like the Lean approach, the annotation process is guided by a set of rules. The program is incrementally annotated with invariant relationships that hold between program variables at intermediate points. Unlike our approach program annotation process is viewed strictly as discovery of invariants. By automating this process the authors meant applying their inference rules in a fashion that does not require significant laborious intellectual efforts and creativity.

A comprehensive study of program annotations is given in [18]. This paper presents a classification of the assertions that were most effective at detecting faults and describes experience using an assertion processing tool that addresses ease-of-use and effectiveness

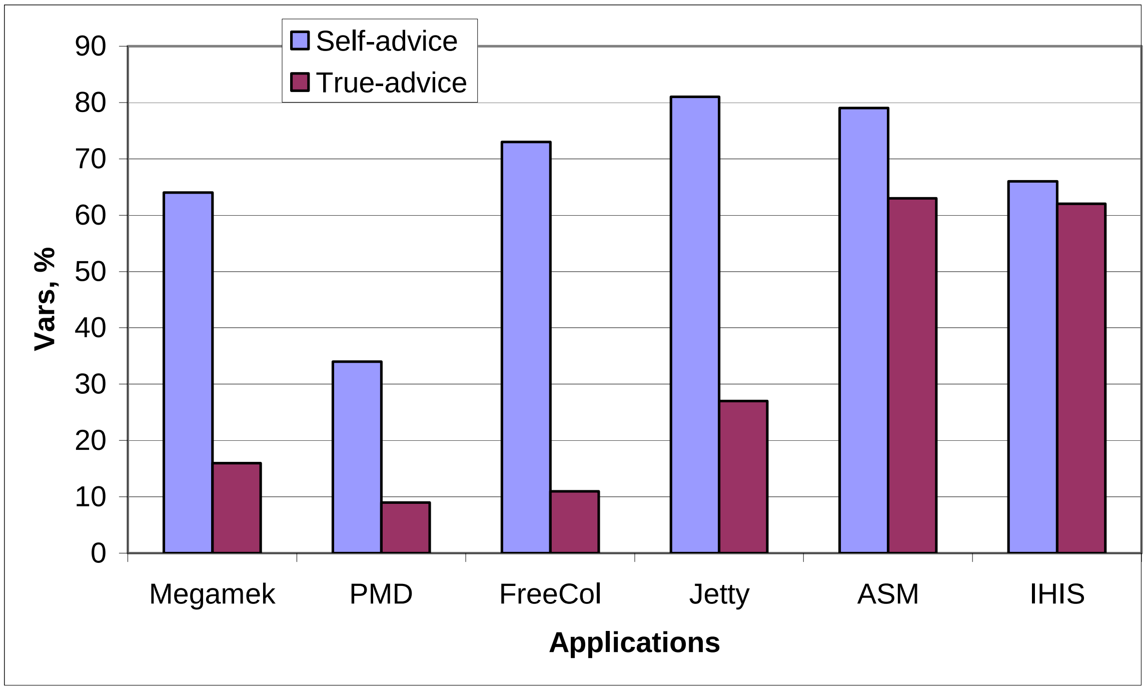


Figure 12. Percentage of variables that the Lean Learner annotates with self-advise (left bar) versus the true-advise annotations (right bar) when the Learner is trained on the VMT application.

of program annotations. Unlike our approach, the tool proposed in this paper does not automate the annotation process.

A technique that can be used to annotate source code with syntactic tags in XML format is described in [19]. Parser generator bison is modified to emit annotating XML tags for an arbitrary LALR grammar. This technique was applied to modify the gcc compiler to annotate C, Objective C, C++ and Java programs with XML tags. While this approach is based on a representation of the parse tree and does not have the same semantic richness as other approaches, it does have the advantage of being language independent, and thus reusable in a number of different domains. Like our research, this parse tree approach uses grammars as external semantic relations to guide the automatic annotation of program code. However, this approach is tightly linked to grammars that do not express domain-specific concepts and relations among them. By contrast, our technique operates on semantic relations and diagrams that are not linked to program source code or a grammar of any language. Their goal is to indicate what language statement or expression corresponds to what grammar construct, while our approach deals with annotating program code with arbitrary semantic concepts.

Calpa is a tool for automating selective dynamic compilation [20]. Calpa's selective dynamic compilation systems is driven by annotations that identify run-time constants, and it can achieve significant program speedups. Calpa is a system that generates annotations automatically for the DyC dynamic compiler by combining execution frequency and value profile information with a model of dynamic compilation cost to choose run-time constants and other dynamic compilation strategies. Calpa is shown to generate annotations of the same or better quality as those found by a human, but in a fraction of the time.

A method for deriving path and loop annotations automatically for object-oriented real-time programs is presented in [21]. Such annotations are necessary when the worst case execution time of programs should be calculated. Normally these annotations must be given manually by the programmer.

A system called Daikon for automatic inferences of program invariants is based on recording values of program variables at runtime with their following analysis [22]. Typically, print statements are inserted in C source code to record the values of parameters to functions and their variables before and after functions are called. Then, these values are analyzed to find variables whose values are not changed throughout the execution of certain functions. These variables constitute invariants that annotate respective functions.

Like our research, Daikon, Calpa and the method proposed in [21] automate the generation of annotations and the user is relieved from a task that can be quite difficult and highly critical. Rather than identifying run-time constants and low-level code properties that are extracted from the source code, our approach enables programmers to automate the process of annotating programs with arbitrary semantic concepts.

An interesting approach in automating text annotations is presented by the OpenText.org project [23]. It is a web-based initiative to develop annotated Greek texts and tools for their analysis. Texts are annotated with various levels of linguistic information, such as text-critical, grammatical, semantic and discourse features. The project offers an annotation tools that helps the user to annotate chunks of text selected by the user with the semantic case roles. The result of the annotation is kept in an XML format which is later converted in a format required by a machine learning program (i.e. Weka). Like in Lean, machine learning algorithms are used to classify text and assign annotations based on the results of the classification. The major difference between our approach and opentext.org is that the latter is used to annotate texts while the former annotates program code.

The rapid explosion in the amount of biological data being generated worldwide is surpassing efforts to manage analysis of the data. As part of an ongoing project to automate and manage bioinformatics analysis, the authors have designed and implemented a simple automated annotation system [24]. The system is applied to existing GenBank/DDBJ/EMBL entries and compared with existing annotations to illustrate not only potential errors but also that they are generally not up-to-date, as a result of new versions of analysis tools and updates of genomic repositories.

The problem of annotating interfaces of object-oriented application frameworks is studied in [25]. Since frameworks provide an established way of reusing the design and implementation of applications in a specific domain its correct usage is important. However, using a framework for creating applications is not a trivial task, however, and special tools are needed for supporting the process. Tool support, in turn, requires explicit annotations of the reuse interfaces of frameworks. Unfortunately these annotations typically become quite extensive and complex for non-trivial frameworks. In this paper the authors focus on describing techniques for minimizing the work needed for creating framework annotations. They discuss the possibility of generating annotations based on frameworks' and example applications' source code, automating annotation creation with dedicated wizards, and introducing coding conventions and advanced language features, such as inheritance, for framework annotations languages. They also introduce a programming environment that supports framework annotation and specialization.

A technique for automatically deriving traceability relations between parts of a specification and parts of synthesized programs is described in [26]. The technique is very lightweight and may work for any process in which one artifact is automatically derived from another. The generality of the technique is illustrated by applying it to two kinds of automatic generation: synthesis of Kalman Filter programs from specifications using the AUTOFILTER program synthesis system, and generation of assembly language programs from C source code using the GCC C compiler. This work is related to our algorithms that enable validation of determined annotations.

The use of semantic annotation in the biochip domain is presented in [27]. They propose a semi-automatic method using the information extraction (IE) techniques for facilitating the generation of ontology-based annotations for scientific articles. The authors evaluate and discuss their method by applying it to the annotation of textual corpus provided by biologists working in the biochip domain. Finally, the authors show that ontology based semantic annotation can improve information retrieval.

10. Discussion and Future Work

One documented problem with WEKA is that it is slow especially with large data sets. As a consequence we do not measure the performance of Lean and plan to address this problem in the future. Clearly, collecting all values of all variables may be prohibitive because it is time consuming and it affects the performance of the program. Currently, we provide a means for programmers to specify variables that should be excluded from monitoring. Eventually, we plan to expand the Lean with an efficient monitoring framework [28] that collects values of selected variables continuously with approximately 3% overhead.

11. Conclusions

We present a novel approach for automating and validating program variables and types with semantic annotations. Our system, called Lean, uses a combination of run-time monitoring, program analysis, and machine-learning approaches to discover and validate annotations for unannotated types and variables based on few initial mappings provided by programmers. We evaluate our approach on open-source software projects. Our results show that after users annotate approximately 6% of the program variables and types, Lean correctly annotates an additional 69% of variables in the best case, 47% in the average, and 12% in the worst case. We also show that true-advice annotation is possible by training the Learner on few programs and applying it to annotate other programs.

References

[1] Joseph W. Davison, Dennis Mancl, William F. Opdyke: Understanding and addressing the essential costs of evolving systems. Bell Labs Technical Journal 5(2): 44-54 (2000)

[2] Albert Endres and Dieter Rombach, A Handbook of Software and Systems Engineering: Empirical Observations, Laws, and Theories. Addison Wesley; May 2003.

[3] http://vmt.sourceforge.net/

[4] Feature-Oriented Domain Analysis (FODA) Feasibility Study [CMU/SEI-90-TR-21]

[5] Krzysztof Czarnecki and Ulrich W. Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, June 2000

[6] AnHai Doan, Pedro Domingos, Alon Y. Halevy: Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. SIGMOD Conference 2001

[7] AnHai Doan, Pedro Domingos, Alon Y. Halevy: Learning to Match the Schemas of Data Sources: A Multistrategy Approach. Machine Learning 50(3): 279-301 (2003)

[8] R. Michalski and G. Tecuci. Machine Learning: A Multistrategy Approach. Morgan Kaufmann, 1994.

[9] W. Cohen and H. Hirsh. Joins that generalize: text classification using Whirl. Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining, 1998.

[10] Tom M. Mitchell, Machine Learning. McGraw-Hill, March 1997.

[11] O. Duda, David G. Stork, and Peter E. Hart. Pattern Classification, Wiley, 2nd edition, November 2001.

[12] Robert Morgan, Building an optimizing compiler, Butterworth-Heinemann, 1998.

[13] Edison Design Group. http://www.edg.com

[14] Ian H. Witten and Eibe Frank. Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann, 2nd edition, June 2005.

[15] http://www.cs.waikato.ac.nz/~ml/weka/index.html

[16] C. A. R. Hoare: The Verifying Compiler: A Grand Challenge for Computing Research. 262-272 Compiler Construction, 12th International Conference, CC 2003, Warsaw, Poland, April 7-11, 2003.

[17] Nachum Dershowitz and Zohar Manna, Inference rules for program annotation.
Third International Conference on Software Engineering, Atlanta, 1978, p. 158 - 167

[18] David S. Rosenblum, A Practical Approach to Programming with Assertions, IEEE Transactions on Software Engineering, vol. 21, no. 1, Jan. 1995, pp. 19-31

[19] James F. Power and Brian A. Malloy: Program Annotation in XML: A Parse-Tree Based Approach. WCRE 2002: p.190

[20] Mock, M. Chambers, C. Eggers, S.J. Calpa: A Tool for Automating Dynamic Compilation, 33rd Annual IEEE/ACM International Symposium on Microarchitecture, 2000. MICRO-33, pp. 291-302.

[21] Jan Gustafsson and Andreas Ermedahl, Automatic derivation of path and loop annotations in object-oriented real-time programs. In Engineering of distributed control systems, p. 81 – 98, 2001

[22] Michael D. Ernst, Jake Cockrell, William G. Griswold, David Notkin: Dynamically Discovering Likely Program Invariants to Support Program Evolution. ICSE 1999: 213-224

[23] http://www.opentext.org

[24] Kim Carter and Akira Oka, Bioinformatics Issues for Automating the Annotation of Genomic Sequences, Genome Informatics 12: 204–211 (2001)

[25] Antti Viljamaa and Jukka Viljamaa, Creating Framework Specialization Instructions for Tool Environments, The Tenth Nordic Workshop on Programming and Software Development Tools and Techniques, Denmark, August 18-20, 2002

[26] Julian Richardson and Jeff Green, Automating Traceability for Generated Software Artifacts, IEEE Conference on Automated Software Engineering (ASE), Linz, Austria, September 20-24, 2004

[27] Khaled Khelif and Rose Dieng-Kuntz Ontology-Based Semantic Annotations for Biochip Domain, Workshop on Knowledge Management and Organizational Memories ECAI 2004, August 2004

[29] Matthew Arnold, Barbara G. Ryder: A Framework for Reducing the Cost of Instrumented Code. PLDI 2001: 168-179