Privacy in Data Mining Using Formal Methods

Stan Matwin[1,2], Amy Felty[1], István Hernádvölgyi[3], and Venanzio Capretta[4]

[1] SITE, University of Ottawa, Canada, {stan,afelty}@site.uottawa.ca
[2] Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
[3] Siemens PSE, Hungary, istvan.hernadvolgyi@siemens.com
[4] Department of Mathematics and Statistics, University of Ottawa, Canada
venanzio.capretta@mathstat.uottawa.ca

Abstract. There is growing public concern about personal data collected by both private and public sectors. People have very little control over what kinds of data are stored and how such data is used. Moreover, the ability to infer new knowledge from existing data is increasing rapidly with advances in database and data mining technologies. We describe a solution which allows people to take control by specifying constraints on the ways in which their data can be used. User constraints are represented in formal logic, and organizations that want to use this data provide formal proofs that the software they use to process data meets these constraints. Checking the proof by an independent verifier demonstrates that user constraints are (or are not) respected by this software. Our notion of "privacy correctness" differs from general software correctness in two ways. First, properties of interest are simpler and thus their proofs should be easier to automate. Second, this kind of correctness is stricter; in addition to showing a certain relation between input and output is realized, we must also show that only operations that respect privacy constraints are applied during execution. We have therefore an intensional notion of correctness, rather that the usual extensional one. We discuss how our mechanism can be put into practice, and we present the technical aspects via an example. Our example shows how users can exercise control when their data is to be used as input to a decision tree learning algorithm. We have formalized the example and the proof of preservation of privacy constraints in Coq.

1 Introduction

Privacy is one of the main concerns expressed about modern computing, especially in the Internet context. People and groups are concerned by the practice of gathering information without explicitly informing the individuals that data about them is being collected. Oftentimes, even when people are aware that their information is being collected, it is used for purposes other than the ones stated at collection time. The last concern is further aggravated by the power of modern database and data mining operations which allow inferring, from combined data sets, knowledge of which the person is not aware, and would have never consented to generating and disseminating. People have no ownership of their own data: it is not easy for someone to exclude themselves from, e.g. direct marketing campaigns, where the targeted individuals are selected by data mining models.

This state of affairs has been amply observed by the legal community, particularly by the segment of it interested in human rights [EPI05]. One of the main concepts that has emerged from research on societal and legal aspects of privacy is the idea of Use Limitation Principle (ULP). That principle states that the data should be used only for the explicit purpose for which it has been collected. It has been noted, however, that "...[ULP] is perhaps the most difficult to address in the context of data mining or, indeed, a host of other applications that benefit from the subsequent use of data in ways never contemplated or anticipated at the time of the initial collection." [IPC98]. A special case of the ULP is the principle of opting out vs. opting in: in most cases one needs to limit explicitly the access to one's data: this approach is called "opting-out". It is widely felt (e.g. [Rie01]) that a better approach would be opting-in, where data could only be collected with an explicit consent for the collection and specific usage from the data owner.

In this paper, we propose and prototype a novel approach which gives an individual the *ownership* of her data: a person may express permissions stating the purposes for which the data may or may not be used. We show a mechanism by which such permissions can be reinforced in a data mining environment. The core of this approach is the use of formal methods for proving properties of programs. We use a theorem prover with a highly expressive logic – the Coq Proof Assistant [Coq03]. This system provides a high degree of power and flexibility for constructing proofs and it is widely used to develop formal proofs of correctness of software. We are able to express data mining programs directly in the logic of the theorem prover, and express privacy properties easily. In the spectrum from less formal to more formal, this kind of system is on the formal end, meaning that the method is more rigorous than many others and thus can provide a higher degree of assurance of correctness than less formal methods. This high degree of assurance is not without cost, of course; the price that must be paid for it is that more work must be done to apply such methods. Proofs can be difficult to construct and require a high degree of interaction and knowledge on the part of the user. We address this issue by modularizing our programs and proofs. In particular, we structure the code so that for each data mining algorithm we consider, we filter out the difficult part of the proof so that it can be done once and for all by an expert, and isolate the code that is likely to change so that the lemmas that are required for this code are straightforward and easy to prove. It should be possible to simplify the task of proving such lemmas even further by exploiting the similarities of such lemmas, and designing algorithms to help automate their proofs.

One aspect that sets privacy verification apart from customary algorithm correctness is that privacy concerns put constraints on the operation of the program. A traditional correctness requirement states a relation between the input and the output of an algorithm and verification consists of proving that the particular software realizes this relation. In our case, the requirement is not on the input-output relation but on the operations that the algorithm performs while running: we impose that no privacy violating operation can be executed. Traditionally, two programs are considered logically equivalent if they implement the same input-output relation; therefore if one of the two satisfies a specification, so does the other. However, we want to discriminate programs on the basis of *how* they process their input into output. In order to realize such discrimination while at the same time preserving the classical logical understanding of functions, we decided to overload the output produced by the program so that a trace of the potentially privacy-breaking operations is preserved in the result.

More specifically, we start from the Weka repository of Java code which implements a variety of data mining algorithms [WF99]. We modify this code to include checks that the privacy constraints that we allow users to specify are met. We also restructure the code to help facilitate proving properties of it. We write it in the functional programming language of Coq, taking care to ensure that the part of the code which checks that users' privacy constraints are met is clearly identifiable. It is this part of the code that we need the flexibility to change. In particular, we want to consider a possibly large variety of privacy constraints, and so we must be able to modify the code as we modify or add new constraints. In general, the lemmas required of this added privacy-checking code will be considerably simpler than lemmas that will be needed about the algorithm as a whole.

This work extends earlier work, which outlined the main ideas and began with a simple algorithm as an example [FM02]. This first example was a program to perform a database join operation, and accommodated users who requested that their data not be used in such an operation. Here we consider a significantly more complex algorithm – a decision tree learning algorithm, we illustrate our method of structuring programs and proofs to tackle the complexity of using formal methods, and we provide a deeper analysis of the issues that arise in making this work practical.

The remainder of this paper is organized as follows: we describe the architecture of our approach, show that it has the desired properties, illustrate how it applies to the learning of decision trees, present the formalization of a particular privacy property, and discuss the acceptance and implementation of our general approach.

2 Architecture

In order to describe the architecture of our approach, let us introduce the players that participate in privacy-conscious data mining:

User C is a consumer or a citizen wishing to state her permissions with respect to her data as it is involved in different data mining processes. Specifying permissions could be as simple as choosing options, both positive and negative, from some fixed set. Data miner Org is an organization involved in processing the data about a number of Cs. D denotes the database schemata of the databases representing that data, while A denotes a set of data mining algorithms that Org runs on the data. B denotes the binaries of the software implementation of A. Data mining software developer Dev develops software S (source code) implementing A. Dev provides Orgs with B. $Veri$ is an independent, generally trusted organization that verifies that C's permissions are respected by Org in the course of the normal operation of Org. Observe that no single player owns all the data.

Our main idea is as follows. User C sets permissions $P_C(D, A)$: what can and cannot be done with her data D by an algorithm in A. Any claim that software S respects these permissions can be stated as a theorem $T(P_C, S)$ about S. Proof $R(P_C, S)$ of this theorem can be checked: if the proof holds, then the program S has the property of respecting P_C. $Veri$ checks both that $R(P_C, S)$ is a proof of $T(P_C, S)$, and that the binary software B run by Org is a compiled form of S. (For example, $Veri$ could compare the hashed result of compilation of S with hashed B, so that $Veri$ needs no access to B, just to its hashed form.)

Graphically, we present this architecture in Fig. 1. Arrows pointing from within a

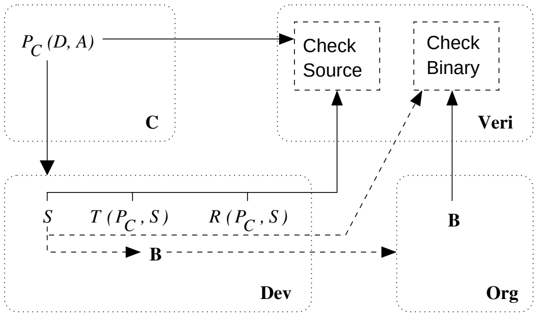

Fig. 1. Architectural diagram of the proposed method

box representing player X to a box representing player Y show that X makes an object at the beginning of the arrow available to Y. For instance, an organization on behalf of consumers makes the set of permissions $P_C(D, A)$ from which each individual C can make choices available to Dev and $Veri$. The dashed line between S and B represents the verifiable link that B is the executable of S. As can be seen from this diagram, the architecture has the following properties:

– The user decides what is and what is not permitted to happen with the data. In that sense, a user's data belong to her.
– Users' permissions are verifiably enforced: it can be proven that the data mining software respects them (or not). Consequently, it can be proven as well that the declared use of the data is adhered to, as long as Org respects the proposed architecture. In that sense, ULP becomes verifiable as well.
– The scheme is robust against cheating by Dev or Org. Dev cannot present a proof of a theorem other than $T(P_C, S)$ because $Veri$ recreates the statement of this theorem from P_C and S. Org cannot run binaries of anything other than S (about which $Veri$ can verify that it satisfies permissions) as $Veri$ can verify the B is indeed a binary version of that S.

3 Example

The decision tree learning algorithm we use is the basic ID3 algorithm from [Mit97]. Decision trees classify examples by following, for a given example, a path from the root to a leaf. This path is determined by the values of the attributes of a given example. The leaf on that path gives the class of the example. Decision tree induction algorithms, such as ID3, take as input examples described by their attributes; each example comes with its class. The output of a decision tree induction algorithm is a decision tree like the one shown in Fig. 2. Among many possible decision trees consistent with the input, ID3 chooses heuristically the one with the highest expected accuracy on unseen data – this will be the best tree. We began by implementing this algorithm in Java. Similar, though somewhat more complex versions can be found in the Weka code. We also implemented the same algorithm in the functional programming language SML because it is a smaller step to go from SML to Coq. To illustrate, we apply our program to fictitious data about loan applications. In that data, people are represented by (among other things) their earnings to expenses ratio, whether or not they live in a single dwelling, and whether they live in the suburbs or in the inner city. The tree produced by ID3 from this dataset is shown in Fig. 2.

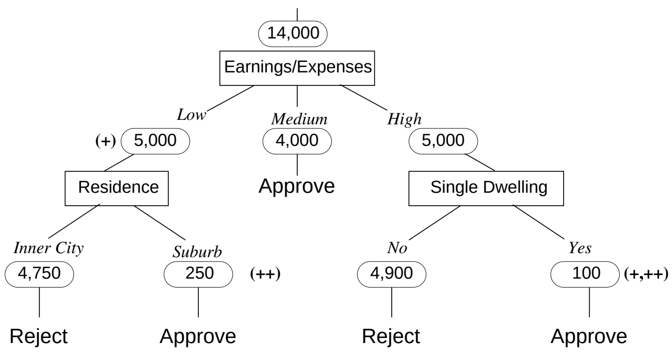

Fig. 2. Example Decision Tree

We have added numbers to each node, indicating the number of training examples considered. For example, the entire training set contains 14,000 records, of which 5,000 are passed on and used to build the tree rooted at the high ratio branch.

Users concerned with privacy may want to restrict how their data is used in training sets to build decision trees. In a typical data mining application the learned decision tree is just a symbolic structure, disconnected from the data used to build its nodes. Business analysts will typically inspect the tree and decide that they are interested in individuals represented by some nodes in the tree, e.g. the rightmost leaf in Fig. 2 (people with high disposable income and likely owning property might be selected for a direct marketing campaign of a life insurance policy, or for a tax audit). Analysts will then perform what is called *data drilling*, i.e. they will request full access to the data subset that resulted in the tree leaf. In such a context, users may want to specify that they do not want their data to be used to build any part of the decision tree unless there is some minimum number of training examples used to build that part of the tree. This may protect them from being singled out or uniquely identified with the condition leading to the leaf in which they find themselves. The ID3 algorithm can be modified to stop building parts of the tree where such constraints are violated. We made this modification, and added two constraints to the data: one of the "single dwelling" loan applicants requires at least 500 people in the training set, and another with the low ratio value requires at least 6,000. The tree resulting from our modified algorithm is the same as Fig. 2 without the branches below the nodes marked with (+). Thus, in the new tree, the low ratio and single dwelling branches will result in no decision. It would be easy, and probably more desirable, to modify the tree so that such branches give some default decision. We leave out this kind of detail when we discuss the proof below.

Data miners might consider such user-imposed constraints to be too restrictive. One alternative is to continue building the tree, and to prune only the leaf node that contains the person whose constraints are violated. Suppose in our example, the person who requires at least 6,000 people in the training set lives in the suburbs. The tree obtained by removing subtrees below nodes marked (++) is the tree obtained from this version of the algorithm. (There is no change to the single dwelling branch in this case.) As before, we will want to modify the two branches that are pruned to give some default decision instead of no decision at all.

These two versions of the algorithm hint at some of the trade-offs and compromises needed between data miners and customers. Instead of discussing this further, we simply note here that (as discussed later), for practical purposes it will be necessary to design a user-friendly language in which users can express constraints. In such a language, we can have a variety of options including the two just discussed here, thus providing increased flexibility.

To obtain an implementation of the ID3 algorithm in Coq's functional language, we began with a direct translation from the SML code and then modified it to use recursion in a style that is more amenable to Coq's reasoning power. Our implementation is modular. We first present a general tree-building procedure that does not depend on specifics such as how to determine the labels of the children of each node or which privacy constraints must be checked. This implementation structure allows us to structure the proof so that we can exploit general mathematical properties of our tree data structures as well as general properties of functions which process such trees.

As mentioned above, we need to overload the output of the algorithm to obtain a trace of the possibly privacy-infringing operations. In the example, the original algorithm produces a search tree that does not contain information about the training data used in its construction. We modify it by labeling each node of the tree with the set of corresponding training data. In this way, the privacy constraints can be verified directly on the output tree. This information can (and should) be discarded before using the tree. This operation can be part of a post-processing phase that also includes replacing branches that give no decision with default decisions as discussed above. We present the algorithm and discuss the formal proof development showing that this code satisfies the required privacy constraints in the next section.

4 The Formal Development

Coq implements the Calculus of Inductive Constructions (CIC), a powerful higher-order logic. In its theory, data types and logical propositions are represented with the same formalism. There are two sorts of types, **Set** for data structures and **Prop** for propositions. An element $A : \mathbf{Set}$ is a type whose terms are elements of the corresponding data structure. An element $P : \mathbf{Prop}$ is a type whose terms are proofs of the corresponding proposition.

The type constructors on $\mathbf{Set}$ (and $\mathbf{Prop}$) are: function types (implication) $A \to B$, with abstraction denoted by $[x : A]b$ and application by $(f\ a)$; binary cartesian products $A \times B$ (conjunction $A \wedge B$); binary disjoint unions $A + B$ (disjunction $A \vee B$); and dependent products $(x : A)B$ (universal quantification $\forall x : A.P(x)$).

Two very important constructs of Coq are inductive and coinductive definitions. The notation to define them is similar, but there are deep differences in their meaning. Below left is the general form of the declaration of an inductive or coinductive type:

(Co)Inductive $A : \mathbf{Set}$	Inductive nat : $\mathbf{Set}$	CoInductive conat : $\mathbf{Set}$
$a_0 : (\Gamma_0)A$	0 : nat	0 : conat
$\vdots$	S : nat $\to$ nat	S : conat $\to$ conat
$a_n : (\Gamma_n)A$		

In the general form, the symbols Γ_i represent sequences of argument assumptions. This declaration introduces a new $\mathbf{Set}$, A; its elements are constructed by applying the constructors a_i to elements satisfying the assumptions Γ_i. The Γ_is can contain occurrences of A itself, provided that they respect a positivity restriction [CP90, PM93]. In this case the elements of A can be constructed recursively. For example, the set nat defined in the middle above contains natural numbers represented as 0, $(S\ 0)$, $(S\ (S\ 0))$, etc.

Coinductive types are defined in an almost identical way and are subject to the same positivity restrictions [Gim98]: The difference between the two constructs is that the recursive elements of an inductive type must be well-founded, while those of a coinductive type are allowed to be infinitely descending. For example, in the set conat defined on the right above, it is possible to construct a term consisting in a infinite sequence of applications of the S constructor: $(S\ (S\ (S\ \cdots)))$, which is not allowed for nat.

Inductive and Coinductive definitions can be given for elements of $\mathbf{Prop}$ as well; and the definition of (co)inductive dependent types and predicates is allowed.

The definition of the type for trees is one of the delicate points in the development. We have adopted an implementation methodology that consists of adapting the data types to the structure of the algorithms to be verified. This implementation philosophy has proved very effective in previous work [BC01, MM04, BC04] and it greatly simplifies the representation of algorithms and the verification of their properties.

To see how this methodology is applied in our case, consider the following informal description of the algorithm: The input is a database, that is a list of records, d, representing the input training data. If the privacy restriction is not satisfied by d (e.g. d is too small), then the construction of the tree is blocked at this node and the node itself gives a default result. On the other hand, if d satisfies the privacy restriction, a node is created and d is partitioned into subsets that will be used to build the children of that node. The children are determined by choosing the attribute that results in the best classifier. A list of databases is obtained by dividing d into equivalence classes, one for each value of this attribute, and a new branch of the tree is created for each element of this list.

This is a top-down construction: The tree is constructed starting from its root node by specifying the branches at each stage. This is typical of coinductively constructed trees, because we cannot be sure *a priori* that the constructed tree is well-founded (it can be proved *a posteriori* in our case). On the other hand, inductive trees are characterized by a bottom-up construction in which the subtrees have to be defined first and the main tree is built from them.

Therefore, the natural definition of $\mathsf{Tree}(A)$, for any $A : \mathbf{Set}$, would be:

$$\begin{array}{l}\mathsf{CoInductive}\ \mathsf{Tree}(A) : \mathbf{Set}\\ \quad \mathsf{node} : A \to \mathsf{list}(\mathsf{Tree}(A)) \to \mathsf{Tree}(A)\end{array}$$

This says that a tree consists of a node with a label of type A and a list of subtrees. Unfortunately, this definition is rejected by the type system of Coq, because the assumption $\mathsf{list}(\mathsf{Tree}(A))$ does not satisfy the positivity condition. This condition states that the type that we are defining ($\mathsf{Tree}(A)$ in our case) can appear in the assumptions of a constructor only in a positive position, that is, either by itself or as the result type of a functional construction. Here it appears inside the $\mathsf{list}()$ constructor and it is therefore rejected. This in spite of the fact that such a definition is sound.

We worked around this limitation of Coq by an alternative definition:

$$\begin{array}{l}\mathsf{CoInductive}\ \mathsf{Tree}(A) : \mathbf{Set}\\ \quad \mathsf{pure\_node} : A \to \mathsf{Tree}(A)\\ \quad \mathsf{add\_child} : \mathsf{Tree}(A) \to \mathsf{Tree}(A) \to \mathsf{Tree}(A)\end{array}$$

The idea here is that we first generate a node ($\mathsf{pure\_node}\ a$), with label a and no branches, and then add the subtrees one by one using the constructor $\mathsf{add\_child}$ (i.e. ($\mathsf{add\_child}\ c\ t$) adds a new child c to existing tree t). In presenting Coq terms in this section, for readability, we often leave type parameters and arguments implicit. (For example, the parameter $[A : \mathbf{Set}]$ is left out of the above $\mathsf{Tree}(A)$ definition and argument A is left out when we write ($\mathsf{pure\_node}\ a$) instead of ($\mathsf{pure\_node}\ A\ a$).) Clearly, all trees that could be constructed by the previous (rejected) definition can be defined in this new format. We can actually define a function node that performs the task that we required of the constructor in the original definition, and just forget about the roundabout way we defined trees:

$$\begin{array}{l}\mathsf{Fixpoint}\ \mathsf{node}\ [a : A; lt : (\mathsf{list}(\mathsf{Tree}(A)))] : \mathsf{Tree}(A) :=\\ \quad \mathsf{Cases}\ lt\ \mathsf{of}\\ \quad\quad \mathsf{nil} \Rightarrow (\mathsf{pure\_node}\ a)\\ \quad\quad (\mathsf{cons}\ t\ lt') \Rightarrow (\mathsf{add\_child}\ t\ (\mathsf{node}\ a\ lt'))\\ \quad \mathsf{end}\end{array}$$

The keyword $\mathsf{Fixpoint}$ indicates a recursive definition on terms of inductive type. In this case, the function node is defined by recursion on the list lt. The Cases construction analyzes the structure of lt (it is either an empty list, nil, or a non-empty list with head t and tail lt'), and uses the function node recursively on the tail of a non-empty list.

However, now it is possible to construct anomalous trees that did not exist earlier: The constructor $\mathsf{add\_child}$ can be recursively applied infinitely many times to generate a node with infinitely many branches. The existence of these pathological trees does not influence in any way the functioning of the algorithm or the proof of correctness.

For coinductive types, there is a construction for recursive definitions similar to Fixpoint, the operation CoFixpoint. The difference is in the criteria that the definition must satisfy: In a Fixpoint definition the recursive calls must be performed on structurally smaller objects; in a CoFixpoint definition there is no restriction on the recursive calls, but the operation must be *guarded*, that is, it must guarantee that for every input, it generates a term with a constructor at its head. For a formal definition of the syntactic conditions for Fixpoint and CoFixpoint see [Coq93, Gim94].

However, we chose a different (equivalent) way to define recursive functions on a coinductive type. We exploit instead the categorical characterization of coinductive types as terminal coalgebras [Hag87] (for a type-theoretic introduction, see also Chapter 3 of [Cap02]; for a comparison of the two approaches, see [Gim94]). A coalgebra, in our case, is a pair consisting of a set A and a function $f : A \to \mathsf{list}(A)$. Terminality of the coinductive type means that for every coalgebra there exists a function $(\mathsf{coalgebra\_tree}\ f) : A \to \mathsf{Tree}(A)$. Intuitively, given an element $a : A$, the term $(\mathsf{coalgebra\_tree}\ f\ a)$ is the tree with root node labeled by a and subtrees recursively constructed by applying $(\mathsf{coalgebra\_tree}\ f)$ to every element of $(f\ a)$. The operator coalgebra_tree can easily be defined by CoFixpoint:

```
CoFixpoint coalgebra_tree_list : (A → list(A)) → A → list(A) → Tree(A) :=
  [f, a, l] Cases l of
            nil ⇒ (pure_node a)
            (cons b l′) ⇒ (add_child (coalgebra_tree_list f b (f b))
                                     (coalgebra_tree_list f a l′))
          end
Definition coalgebra_tree : (A → list(A)) → A → Tree(A) :=
  [f, a](coalgebra_tree_list f a (f a))
```

Notice that in the CoFixpoint definition of coalgebra_tree_list the recursive calls are performed on arguments that are not necessarily structurally simpler than the original input, but they are *guarded* by the application of the constructor add_child which ensures that the construction of the tree proceeds by at least one step. The Definition keyword introduces a (non-recursive) definition in Coq.

We first define a general decision-tree-building procedure that does not depend on the specific data used by the ID3 algorithm. In particular, we assume the existence of the following parameters with their types, but assume nothing about their implementations: $A : \mathbf{Set}$; $\mathsf{children\_list} : A \to \mathsf{list}(A)$; $\mathsf{constraint} : A \to \mathsf{bool}$; $\mathsf{dummy} : A$. Node labels will be elements of type A. The children_list function, given an input a, will determine the elements of type A that will be used to construct the children of a node labelled with a. The children of a node are uniquely determined by the label, and there are never two nodes with the same label in any decision tree. The constraint predicate identifies the subset of A that satisfies a particular property, left unspecified here. It returns a boolean value true or false. Finally, dummy is an unspecified default value of type A. Given these parameters, Fig. 3 contains the general implementation of decision_tree. The first definition is a general filter function which, given an input

```
Fixpoint filter [z : A; f : A → bool; l : list(A)] : list(A) :=
  Cases l of
    nil ⇒ nil
    (cons a l′) ⇒ if (f a)
                   then (cons a (filter z f l′))
                   else (cons z (filter z f l′))
  end
Definition secure_children A → list(A) :=
  [a](filter A dummy constraint (children_list a))
Definition secure_tree : A → Tree(A) :=
  [a](coalgebra_tree A secure_children a)
Definition decision_tree : A → Tree(A) :=
  [a]if (constraint a) then (secure_tree a) else (pure_node dummy)
```

Fig. 3. The General Decision Tree Algorithm in Coq

list, replaces every element a that doesn't meet the constraint $(f\ a)$ with some default value z. The secure_children function uses this filter function on the list obtained by calling children_list. The function secure_tree calls coalgebra_tree with secure_children as its argument function. Thus we define it by using the characterization of the coinductive type Tree(A) as a terminal coalgebra. The top-level decision_tree function calls secure_tree, but first checks to see that the initial input meets the required constraint. If not, a degenerate tree of one node with a dummy label is returned.

To instantiate the parameters and specialize this algorithm to ID3, we need several definitions. We begin by using Coq's built-in arrays and lists to represent the input training data. A record is represented as an array of fixed size. Each position in the array contains a particular field, and we assume the fields are in the same order in each record in the training data. The training data is a list of such records. Defining an array in Coq requires the type and number of fields. We leave these unspecified here. They are formal parameters to our program, which we call Field and numFields. For simplicity we assume that all fields have the same type (i.e. all possible field contents can be encoded as elements of type Field). We define our array and lists as follows:

```
Definition Record := array(numFields, Field)
Definition DB := list(Record)
```

A privacy constraint is associated with each record. We represent this association as a function min_data : Record → nat. For a record r : Record, (min_data r) specifies the minimum number of records that must be present in a node of the decision tree for the algorithm to be allowed to proceed. This number could, for example, be stored as one of the fields in r, and then min_data would be the function that extracts the value of this field. Fig. 4 contains the implementation of several functions we will need. The first two functions define the code that checks whether privacy is respected for all the records in the database. The first function, min_data_check checks that all records in a certain database have a privacy specification smaller than a given bound. In the second function, privacy_constraint, we just require that the boolean relation min_data_check is satisfied when n is the size of the database.

```
Fixpoint min_data_check [db : DB; n : nat] : bool :=
   Cases db of
      nil ⇒ true
      (cons r db′) ⇒ ((min_data r) ≤ n and (min_data_check db′ n))
   end
Definition privacy_constraint : DB → bool :=
   [db](min_data_check db (length db))
Definition leaf_db : DB → bool :=
   [db](length (partitioning_functions db)) ≤ 1
Definition id3_children : DB → list(DB) :=
   [db]if (leaf_db db) then nil else (partitions db)
```

Fig. 4. Functions Specific to the ID3 Algorithm

We leave out the part of the ID3 algorithm that determines how to partition the training data by choosing the attribute that results in the best classifier. The details have no bearing on the privacy issue. We just assume the existence of the following function:

$$\mathsf{partitioning\_functions} : \mathsf{DB} \rightarrow \mathsf{list}(\mathsf{Record} \rightarrow \mathsf{bool})$$

Given a database db : DB, (partitioning_functions db) returns a list of boolean predicates over Record. Each predicate selects a particular equivalence class.

For our proof, we need no information about this function other than its type. (For example, we do not even need to know that each predicate selects a subset of db that is disjoint from all others.) If there is only one equivalence class, the ID3 algorithm builds a leaf node and stops. The function leaf_db in Fig. 4 tests for this case by checking the length of the list returned by partitioning_functions. When there is more than one equivalence class, we must compute the partitions of db. We can easily define a function that does so according to the predicates returned by partitioning_functions:

$$\mathsf{partitions} : \mathsf{DB} \rightarrow \mathsf{list}(\mathsf{DB})$$

We omit the definition. This function is used by id3_children in Fig. 4, which first tests whether or not this partitioning should be done by calling leaf_db. In the true case, since there will be no children, the empty list is returned.

The ID3 algorithm is then obtained by instantiating the parameters introduced above. First, we instantiate A with DB. Thus, we store at each node the actual subset of the training data used to build the subtree below the node. (Note that our trees do not store labels such as "Approve", "Reject", or "Single Dwelling" as used in Fig. 2. They are not important for constructing or using the tree.) To complete the algorithm, we instantiate children_list with id3_children, constraint with privacy_constraint, and dummy with null_data. We define null_data to be an empty list of training data. This is the label used for nodes which do not meet the required privacy constraints. Let id3_decision_tree be the name of the version of the decision_tree function with parameters instantiated in this way.

To implement the version of the algorithm that keeps more nodes and eliminates only the leaves marked with (++), the only change needed is to instantiate the constraint parameter with the following code:

$$\text{Definition leaf\_privacy\_constraint : DB} \rightarrow \text{bool} :=$$
$$[db](\text{leaf\_db } db) \text{ implies } (\text{min\_data\_check } db \ (\text{length } db))$$

instead of instantiating with privacy_constraint.

The following definition of the predicate privacy_pred expresses what it means for user constraints to be satisfied by source code S.

$$\begin{aligned}&\text{Definition privacy\_pred} := [S : \text{DB} \rightarrow \text{Tree(DB)}]\\&\quad \forall db_0, db_1 : \text{DB}.(\text{In\_tree } db_1 \ (S \ db_0))\\&\quad \rightarrow \forall r : \text{Record}.(\text{In } r \ db_1) \rightarrow (\text{min\_data } r) \preccurlyeq (\text{length } db_1)\end{aligned}$$

The predicate $(\text{In\_tree } db_1 \ (S \ db_0))$ expresses the fact that the database db_1 is the label of a node of the tree generated by S for the training set db_0. The predicate says that if r is one of the records in db_1 then the privacy restriction is satisfied, that is, the number of records in db_1, $(\text{length } db_1)$, is at least the minimum limit specified for r, $(\text{min\_data } r)$. We use the symbol $\preccurlyeq$ for the logical version of the order relation to distinguish it from the boolean version used in the algorithm.

Note here that S is a formal parameter. The theorem that is written $T(P_C, S)$ is obtained in this case by the application (privacy_pred id3_decision_tree). The heart of the proof of this theorem is a series of lemmas showing that $(\text{privacy\_constraint } a) = true$ is an invariant of all nodes a created by the id3_decision_tree program. The main theorem follows fairly directly from this property. The version of the algorithm that uses leaf_privacy_constraint instead of privacy_constraint requires only minor modifications to two lemmas and their proofs. The whole proof development, including definitions, lemmas, and proofs, is roughly 500 lines of Coq script.

5 Discussion and Conclusion

Let us turn our attention to some of the practical aspects of the approach we are proposing. These include the additional effort (human and computational) needed to perform data mining compared to the current practice; the question of access of the players to the information proprietary to other players; and the limitations of the approach.

Firstly, as already mentioned, proving the theorem $T(P_C, S)$ is hard, but this needs to be done only once. We envisage that Dev will perform this as part of the documentation activities. The proof $R(P_C, S)$ must be checked by $Veri$. This check can be performed automatically. Instead of being done exhaustively, it can be done at random times, similarly to industrial quality control. Finally, a computational overhead of the software modified so that permissions are checked during execution of B is linear in the number of Cs whose privacy is checked.

Secondly, let us see in more detail what kind of access different players need to have to software belonging to other players. It is access to S that is difficult in practice: for obvious reasons Devs will be reluctant to let other parties read the source code of Dev's proprietary software. We believe that these concerns can be addressed by carefully analyzing and constraining the access process, and engineering it so that the source code is only accessed by programs and never by humans. For instance, $Veri$ needs access to S when checking proof R, but that can be done in Dev's environment, by an applet or other non-intrusive mechanism for which it is known that it does not export any information outside that environment.

Let us now look at some issues related to the language in which Cs express their permissions of P_C. The first question is the issue of names of database fields – how would C know what names of the database fields are needed to describe her permissions? We can see this answered when universal XML standards will normalize the names of fields in large databases. Alternatively, one can envisage the disclosure of field names by Orgs participating in the proposed scheme.

Finally, the language of P_Cs also limits our approach to data properties that can be expressed syntactically in formal logic. This does not take into account data dependencies that may be true in a given domain and exploited by Orgs that have that domain knowledge. It may be possible to deduce information from decision trees that is not covered by privacy constraints. An example from the real world of deductions from data is *mortgage redlining*. This is a name for a discrimination technique that has been used in the past by some US lenders to exclude mortgage loan applicants based on race and ethnic criteria. Racial redlining has been ruled illegal some years ago, but many (see [USC98]) allege that lenders use other "attributes" of loan applicants that the lenders know correlate highly with race, such as a combination of the geographic info (e.g. ZIP indicating inner city) with household income. This results in the same effect as racial redlining, and shows the limitation of "syntactic" privacy permissions that can be sidetracked by having the knowledge of deep relationships between attributes. This is the case in our example with the loan data and the resulting decision tree in Fig. 2: while grouping people by race may be forbidden by law, lenders may know that following the inner city path in the tree may practically identify racial minorities. In general, it is important that users be given as much information as possible about what their chosen privacy constraints cover, and what they don't.

Most related work on addressing privacy problems in the data mining context [AS00, ESAG02, Iye02] approaches the problem by applying data transformations that perturb values of individual data records, changing the "sensitive" fields (e.g. salary information). While the value of an individual perturbed field becomes useless, a reconstruction procedure estimates the original distribution, so that a modified decision tree induction algorithm gives results close to those that would be obtained on the original, unperturbed database. Another branch of this research looks at the privacy aspects when the data is split either vertically [VC02] (i.e. attributes are partitioned, and one party knows only a given partition and does not wish to share the values of these attributes with other parties, while all attributes are needed for data mining), or horizontally [KC02] (i.e. the database is partitioned into subsets of records, one party knows only the records in its partition and does not wish to share these records with other parties, while all records are needed for data mining). None of this work, however, offers any tools to address the ULP.

A variety of approaches to the privacy problem introduce formal models which can serve as a starting point for verifying privacy policies. One example, which does begin to address the ULP, is a language based approach which builds information-flow into the types of a simplified version of Java [HA04]. Although this work does not address data mining in particular, it may be possible to integrate this kind of approach with ours to improve the scope of privacy concerns that can be enforced.

A wealth of future work is ahead of us. A user-friendly permission language for P_Cs, easy to handle by an average person, needs to be designed. As suggested earlier, it could initially have the form of a set of options from which C would choose her permissions, both negative and positive. Tools for proof development that can ease proof construction in this domain need to be designed. Our current work includes experimenting with the Krakatoa approach [MPMU04] which allows us to work more directly with the Weka Java code, avoiding the step of translating code into Coq. We hope this approach will also provide better automation of proofs. Also, the approach presented here can be combined with the data perturbation method mentioned earlier [AS00]. In our framework, one could prove that the perturbation techniques are in fact applied to the data during data mining. Finally, we need to experiment with a specific dataset used by an organization which will accept to act as the first Org, and a Dev who will provide access to his S on the basis described above.

Acknowledgments

The authors acknowledge the support of the Natural Sciences and Engineering Research Council of Canada, and Communications and Information Technology Ontario. We also thank Guillaume Dufay for useful discussions.

References

[AS00] R. Agrawal and R. Srikant. Privacy-preserving data mining. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *2000 ACM SIGMOD International Conference on Management of Data*, pages 439–450. ACM, May 2000.

[BC01] Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, 2001.

[BC04] Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. To appear in *Mathematical Structures in Computer Science*. Available at http://www.science.uottawa.ca/˜vcapr396/, 2004.

[Cap02] Venanzio Capretta. *Abstraction and Computation*. PhD thesis, Computing Science Institute, University of Nijmegen, 2002.

[Coq93] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs. International Workshop TYPES'93*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1993.

[Coq03] Coq Development Team. The Coq Proof Assistant reference manual: Version 7.4. Technical report, INRIA, 2003.

[CP90] Thierry Coquand and Christine Paulin. Inductively defined types. In P. Martin-Löf, editor, *Proceedings of Colog '88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[EPI05] EPIC. Electronic Privacy Information Center. http://www.epic.org/, 2005.

[ESAG02] A. Evfimievski, R. Srikant, R. Agrawal, and J. Gehrke. Privacy preserving mining of association rules. In *Eighth ACM SIGKDD International Conference on Knowledge Discovery in Databases and Data Mining*, July 2002.

[FM02] Amy Felty and Stan Matwin. Privacy-oriented data mining by proof checking. In *Sixth European Conference on Principles of Data Mining and Knowledge Discovery*, volume 2431 of *Lecture Notes in Computer Science*, pages 138–149. Springer-Verlag, August 2002.

[Gim94] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs. International Workshop TYPES '94*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer-Verlag, 1994.

[Gim98] Eduardo Giménez. A Tutorial on Recursive Types in Coq. Technical Report 0221, Unité de recherche INRIA Rocquencourt, May 1998.

[HA04] Katia Hayati and Martín Abadi. Language-based enforcement of privacy policies. In *Proceedings of Privacy Enhancing Technologies Workshop (PET 2004)*, 2004.

[Hag87] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, volume 283 of *Lecture Notes in Computer Science*, pages 140–157. Springer-Verlag, 1987.

[IPC98] IPCO. Data mining: Staking a claim on your privacy, Information and Privacy Commissioner/Ontario. http://www.ipc.on.ca/scripts/index.asp?action=31&-P_ID=11387&N_ID=1&PT_ID=11351&U_ID=0, January 1998.

[Iye02] Vijay S. Iyengar. Transforming data to satisfy privacy constraints. In *Eighth ACM SIGKDD International Conference on Knowledge Discovery in Databases and Data Mining*, pages 279–287, July 2002.

[KC02] Murat Kantarcioglu and Chris Clifton. Privacy-preserving distributed mining of association rules on horizontally partitioned data. In *The ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'2002)*, June 2002.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[MM04] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[MPMU04] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, January–March 2004.

[PM93] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, 1993. LIP research report 92-49.

[Rie01] D. G. Ries. Protecting consumer online privacy – an overview. http://-www.pbi.org/Goodies/privacy/privacy_ries.htm, May 2001.

[USC98] USCM. Mayors attack urban redlining, mortgage discrimination, The US Conference of Mayors. http://www.usmayors.org/uscm/news/press_releases/press-_archive.asp?doc_id=98, 1998.

[VC02] Jaideep Vaidya and Chris Clifton. Privacy preserving association rule mining in vertically partitioned data. In *Eighth ACM SIGKDD International Conference on Knowledge Discovery in Databases and Data Mining*, July 2002.

[WF99] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.