Classification of Individuals with Complex Structure

A.F. Bowers BOWERS@CS.BRIS.AC.UK
C. Giraud-Carrier CGC@CS.BRIS.AC.UK
Department of Computer Science, University of Bristol, Bristol, BS8 1UB, U.K.

J.W. Lloyd JWL@CSL.ANU.EDU.AU
Computer Sciences Laboratory, Research School of Information Sciences and Engineering, Australian National University, Canberra, ACT 0200, Australia

Abstract

This paper introduces a foundation for inductive learning based on the use of higher-order logic for knowledge representation. In particular, the paper (i) provides a systematic individuals-as-terms approach to knowledge representation for inductive learning, and demonstrates the utility of types and higher-order constructs for this purpose; (ii) introduces a systematic way to construct predicates for use in induced definitions; and (iii) widens the applicability of decision-tree algorithms beyond the usual attribute-value setting to the classification of individuals with complex internal structure. The paper contains several illustrative applications. The effectiveness of the approach is demonstrated by applying the decision-tree learning system to two benchmark problems.

1. Introduction

Traditionally, inductive learners have used the attribute-value language to represent individuals in (supervised) learning from examples. Though the relative simplicity of this attribute-value representation allows the construction of efficient learning systems, it also restricts their applicability, as individuals must be represented by tuples of constants. In some applications, the structure of individuals is too rich to be adequately captured by such a representation.

What follows is an introduction to a new foundation for inductive learning that naturally extends the attribute-value framework to account for applications where the individuals to be classified have complex internal structure. We use a typed, higher-order logic as the knowledge representation formalism. This logic provides highly expressive hypothesis languages, supporting a variety of data types, including sets, multisets, and graphs. Induced definitions in such hypothesis languages are comprehensible, thus providing insight into the nature of the application data. We employ the Escher programming language (Lloyd, 1999), a higher-order declarative programming language that extends the functional language Haskell, as the embodiment of a computationally useful subset of this logic. This paper provides an introduction to the main ideas; two other papers in preparation provide further details on predicate construction, the decision-tree learning system, and the theoretical underpinnings.

2. Representation of Individuals

In an inductive learning problem, there is some collection of individuals for which a general classification is required. In the context of attribute-value learning, these individuals are represented by tuples of constants. In the general setting, more complex types are available for representing individuals. Examples are available which state the class of certain individuals. The classification will be given by a function from the domain of the individuals to some small finite set corresponding to the classes. Note that the reformulation of attribute-value learning presented here is, in a sense, "isomorphic" to the traditional one which uses the attribute-value language and thus, if one confines attention purely to the attribute-value setting, nothing much has been gained. The primary advantage of our formulation is that it *generalizes* directly to the representation of individuals with complex internal structure.

We adopt a standard approach to knowledge representation. The basic principle is that *an individual should be represented by a (closed) term*; we refer to this as the 'individuals-as-terms' approach. For a complex individual, the term will be correspondingly complex. Nevertheless, this approach has significant advantages: the representation is compact, all information about an individual is contained in one place, and the structure of the term provides strong guidance on the search for a suitable induced definition.

What types are needed to represent individuals? Typically, one needs the following: integers, floats, characters, strings, and booleans; data constructors; tuples; sets; multisets; lists; trees; and graphs. The first group are the basic building blocks of terms representing individuals. In Escher, the type of integers is denoted by `Int`, the type of floats by `Float`, and the type of booleans by `Bool`. Also needed are data constructors for user-defined types. For example, see the data constructors `Abloy` and `Chubb` for the type `Make` in Section 4 below. Tuples are essentially the basis of the attribute-value representation of individuals, so their utility is clear. Less commonly used elsewhere for representing individuals are sets and multisets. However, sets, especially, and multisets are basic and extremely useful data types, and we make much use of them. The Escher language provides a rich collection of higher-order functions for manipulating sets and multisets. Other constructs needed for representing individuals include the standard data types, lists, trees, and graphs. This catalogue of data types is a rich one, and intentionally so. We advocate making a careful selection of the type which best models the application being studied.

3. Predicate Construction

The problem we face is to induce a suitable definition for the desired target function, assuming that the individuals in the domain of this function are represented by (possibly complex) terms. For this, we need to investigate the detailed structure of these terms to construct predicates that the individuals may or may not satisfy. At this point the accurate modelling pays off, as *the type selected to model individuals strongly suggests the predicates that will be useful for classifying individuals*. However, even with the very strong hints provided by the types, finding a suitable predicate is a search problem – one must search in a systematic way through a space of potential predicates. This search space is made up by composing more basic functions into predicates in all possible ways. So we need to say something about what these more basic functions are and how they are composed.

First, we deal with composition. This is simply handled by the function '`.`' having the signature `(R -> S) -> (S -> T) -> (R -> T)` and defined by `(f.g) x = g (f x)`, where `R`, `S` and `T` are types.

Next we turn to the appropriate set of functions. A *transformation* is a function `f` having the signature `(R1 -> Bool) -> ... -> (Rk -> Bool) -> S -> T`, where `R1`, ..., `Rk`, `S` and `T` are types and $k \geq 0$. The idea is that, if `ri :: Ri -> Bool`, `i` = `1`, ... , `k`, are predicates, then `(f r1 ... rk)` is a function of type `S -> T` and functions obtained like this can be composed to obtain a predicate. A special case is when $k = 0$, in which case the transformation is itself such a suitable function.

We now introduce some useful transformations. Given several predicates, it is common to want to construct the predicate obtained by forming the "conjunction" of these predicates. This is achieved via the transformation `andN` (where $N \geq 2$) having the signature `(T -> Bool) ->...-> (T -> Bool) -> T -> Bool` and defined by `andN p1 ... pN x = (p1 x) && ... && (pN x)`. If `N` is an integer, then `(==N) :: Int -> Bool` is a transformation, in fact a predicate, defined by `(==N) m = m == N`. Similarly, one can define transformations `(<N)` corresponding to `<`, `(>N)` corresponding to `>`, `(/=N)` corresponding to `/=`, and so on. More generally, if `T` is a type and `C` a constant of type `T`, then we can make a predicate `(==C)` on `T` by defining `(==C) x = x == C`. Similarly, the predicate `(/=C)` on `T` is defined by `(/=C) x = x /= C`. For any type `T`, there is a transformation `top :: T -> Bool` defined by `top x = True`.

We now discuss three types more systematically, giving various transformations and predicates defined on them. In the following, the keyword `data` indicates the declaration of a type and the data constructors of that type, and `type` indicates a type synonym.

3.1 Tuples

The transformations for tuples are the projections `projTi :: (T1, ..., Tn) -> Ti` defined by `projTi (t1, ..., tn) = ti`, for `i = 1, ..., n`.

As an example of predicate construction, suppose `C` is a constant of type `T1`. Then, by composition, we can construct the predicate `projT1.(==C)` on individuals of type `(T1, ... , Tn)`. This predicate is true on an individual of this type iff its first component is `C`.

3.2 Sets

In a higher-order logic and in Escher, a set is identified with a predicate, its characteristic function. Thus a set whose elements have type `T` is a function of type `T -> Bool`. When we wish to informally make a distinction between sets and predicates (even though mathematically they are identical), we use the synonym `{T}` for `T -> Bool`. Analogously, a multiset whose elements have type `T` is a function of type `T -> Int`, where the value of the function at some element is the multiplicity of the element, that is, the number of times it appears in the multiset.

For sets of type `{T}`, consider the transformation `setExists1 :: (T -> Bool) -> {T} -> Bool` defined by `setExists1 b t = exists \x -> (b x) && x 'in' t`. Then, if `b` is a predicate on the type `T`, `(setExists1 b)` is a predicate of type `{T} -> Bool` that is true of a set iff it has an element which satisfies the predicate `b`. One can also define `setExists2` which looks for two elements satisfying given conditions, and so on. Analogously, for multisets, one has `msetExists2`.

Now consider the transformation

```
domCard :: (T -> Bool) -> {T} -> Int;
domCard b t = card {x | (b x) && x 'in' t};
```

where `card` computes the cardinality of a set. Thus, for each predicate `b`, the function `(domCard b) :: {T} -> Int` computes the cardinality of the subset of elements of the argument that satisfy the predicate `b`.

3.3 Graphs

For (undirected) graphs, there is a type constructor `Graph` such that the type of a graph is `(Graph V E)`, where `V` is the type of information in the vertices and `E` is the type of information in the edges. `Graph` is defined as follows.

```
type Label = Int;
type Graph V E = ({(Label, V)},
                        {(Label -> Int, E)});
```

To avoid the need for explicit references to the label type, we introduce a type constructor `Vertex` such that the type of a vertex is `(Vertex V E)` and a type constructor `Edge` such that the type of an edge is `(Edge V E)`. Here are some transformations on graphs.

```
vertices :: Graph V E -> {Vertex V E};
edges :: Graph V E -> {Edge V E};
vertex :: Vertex V E -> V;
edge :: Edge V E -> E;
connects :: Edge V E -> (Vertex V E -> Int);
(subgraphs N) :: Graph V E -> {Graph V E};
```

The transformation `vertices` returns the set of vertices of a graph, `edges` returns the set of edges of a graph, `vertex` returns the information at a vertex, `edge` returns the information on an edge, `connects` returns the (unordered) pair of vertices joined by an edge, and `(subgraphs N)` returns the set of (connected) subgraphs of the graph that contain `N` vertices.

4. Two Easy Pieces

We now present two simple applications which illustrate the ideas that have been introduced.

4.1 Sets

Consider the problem of determining whether a key in a bunch of keys can open a door. More precisely, suppose there are some bunches of keys and a particular door which can be opened by a key. For each bunch of keys either no key opens the door or there is at least one key which opens the door. For each bunch of keys it is known whether there is some key which opens the door, but it is not known precisely which key does the job, or it is known that no key opens the door. The problem is to find a classification function for the bunches of keys, where the classification is into those which contain a key that opens the door and those that do not. This problem, a multiple-instance problem, is prototypical of a number of important practical problems such as drug activity prediction (Dietterich et al., 1997). We make the following declarations.

```
data Make = Abloy | Chubb | Rubo | Yale;
data Length = Short | Medium | Long;
data Width = Narrow | Normal | Broad;
type NumProngs = Int;
type Key = (Make, NumProngs, Length, Width);
type Bunch = {Key};
```

We want to learn the function `opens :: Bunch -> Bool`. Here is a typical example.

```
opens {(Abloy, 4, Medium, Broad),
       (Chubb, 3, Long, Narrow),
       (Abloy, 3, Short, Normal)} = True;
```

The hypothesis language contains the following transformations.

```
(==Abloy) :: Make -> Bool;
   ...
(==Broad) :: Width -> Bool;
projMake :: Key -> Make;
projNumProngs :: Key -> NumProngs;
projLength :: Key -> Length;
projWidth :: Key -> Width;
setExists1 :: (Key -> Bool) -> Bunch -> Bool;
```

together with `and2`, `and3` and `and4` on the type `Key`. Our learning system (see Section 5) found the following definition for the function `opens`.

```
opens b =
  if setExists1 (and2 (projMake.(==Abloy))
                  (projLength.(==Medium))) b
  then True
  else False;
```

"A bunch of keys opens the door if and only if it contains an Abloy key of medium length".

4.2 Graphs

The next application involves learning a theory to predict whether a chemical molecule has a certain property. We use an undirected graph to model a molecule – an atom is a vertex in the graph and a bond is an edge. The type `Element` is the type of the (relevant) chemical elements.

```
data Element = Br | C | Cl | ... | N | O | S;
```

We also make the following type synonyms.

```
type AtomType = Int;
type Charge = Float;
type Atom = (Element, AtomType, Charge);
type Bond = Int;
```

We now obtain the type of a molecule which is an (undirected) graph whose vertices have type `Atom` and whose edges have type `Bond`, which leads to the following definition.

```
type Molecule = Graph Atom Bond;
```

We want to learn the definition of the function `active :: Molecule -> Bool`. A typical example is as follows (where `<n,m>` denotes an unordered pair).

```
active ({(1, (C, 22, -0.117)),
               ...
         (12, (C, 27, 0.013))},
        {(<1,2>, 7), (<1,6>, 7), (<2,3>, 7),
               ...
         (<6,8>, 2), (<11,12>, 7)}) = True;
```

The hypothesis language contains the following transformations.

```
(==Br) :: Element -> Bool;
   ...
(==S) :: Element -> Bool;
(==3) :: AtomType -> Bool;
   ...
(==195) :: AtomType -> Bool;
(<=-0.117) :: Charge -> Bool;
   ...
(>=0.142) :: Charge -> Bool;
(==1) :: Bond -> Bool;
   ...
(==7) :: Bond -> Bool;
(>0) :: Int -> Bool;
(>1) :: Int -> Bool;
(>2) :: Int -> Bool;
projElement :: Atom -> Element;
projAtomType :: Atom -> AtomType;
projCharge :: Atom -> Charge;
vertices :: Molecule -> {Vertex Atom Bond};
edges :: Molecule -> {Edge Atom Bond};
vertex :: Vertex Atom Bond -> Atom;
edge :: Edge Atom Bond -> Bond;
connects :: Edge Atom Bond ->
                  (Vertex Atom Bond -> Int);
domCard :: (Vertex Atom Bond -> Bool) ->
                  {Vertex Atom Bond} -> Int;
domCard :: (Edge Atom Bond -> Bool) ->
                    {Edge Atom Bond} -> Int;
msetExists2 :: (Vertex Atom Bond -> Bool) ->
   ... -> (Vertex Atom Bond -> Int) -> Bool;
setExists1 :: (Molecule -> Bool) ->
                         {Molecule} -> Bool;
(subgraphs 3) :: Molecule -> {Molecule};
```

together with several conjunction transformations. Our learning system found the following definition for the function `active` (demonstrating it could find a complicated concept that we had put into the data).

```
active m =
 if (subgraphs 3).(setExists1 (and2
     (vertices.(domCard
          (vertex.projAtomType.(==38))).(>0))
     (edges.(domCard (and2
       (connects.(msetExists2
           (vertex.projAtomType.(==40)) top))
       (edge.(==2)))).(>1))))
 then True
 else False;
```

"If a molecule contains 3 atoms connected by bonds such that at least one atom has atom type 38, and at least two edges connect an atom having atom type 40 and also have bond type 2, then it is active; else it is not active".

5. Decision-Tree Algorithm

We now outline a decision-tree learning algorithm based on these ideas.

5.1 A Pruning Theorem

First we give a pruning theorem that the learner uses to safely prune the search space of predicates. Suppose there are c classes in all. Let $\mathcal{E}$ be a (non-empty) set of examples, N the number of examples in $\mathcal{E}$, n_i the number of examples in $\mathcal{E}$ in the ith class, and $p_i = n_i/N$, for $i = 1, \ldots, c$.

Definition We define the *accuracy*, $A_{\mathcal{E}}$, of the set $\mathcal{E}$ of examples by $A_{\mathcal{E}} \equiv p_M$, where M is the index of the majority class of $\mathcal{E}$.

Definition Let $\mathcal{P} = \{\mathcal{E}_1, \ldots, \mathcal{E}_m\}$ be a partition of a set of N examples, where there are N_j examples in $\mathcal{E}_j$, for $j = 1, \ldots, m$. Then we define the *accuracy*, $A_{\mathcal{P}}$, of the partition $\mathcal{P}$ by $A_{\mathcal{P}} \equiv \sum_{j=1}^{m} \frac{N_j}{N} A_{\mathcal{E}_j}$

Our decision-tree learning algorithm makes binary splits at each node in the tree, so we develop some material about binary partitions. If $\mathcal{E}$ is a set of examples, then a predicate b induces a partition $\{\mathcal{E}_1, \mathcal{E}_2\}$ of $\mathcal{E}$, where $\mathcal{E}_1$ is the set of examples that satisfy b and $\mathcal{E}_2$ is the set of examples that do not satisfy b. Now a rewrite (see Section 5.2) takes a predicate b and constructs a new predicate b' by strengthening b. Thus the new predicate b' implies b and the partition $\{\mathcal{E}_1', \mathcal{E}_2'\}$ of $\mathcal{E}$ induced by b' has the property that $\mathcal{E}_1' \subseteq \mathcal{E}_1$, where $\mathcal{E}_1'$ is the set of examples that satisfy b'. These considerations lead to the following definition.

Definition Let $\mathcal{E}$ be a set of examples and $\{\mathcal{E}_1, \mathcal{E}_2\}$ a partition of $\mathcal{E}$. We say a partition $\{\mathcal{E}_1', \mathcal{E}_2'\}$ of $\mathcal{E}$ is a *refinement* of $\{\mathcal{E}_1, \mathcal{E}_2\}$ if $\mathcal{E}_1' \subseteq \mathcal{E}_1$.

Next we introduce a measure for partitions that is crucial for pruning the search space of predicates.

Definition Let $\mathcal{P} = \{\mathcal{E}_1, \mathcal{E}_2\}$ be a partition of a set $\mathcal{E}$ of N examples, where n_i is the number of examples in $\mathcal{E}$ in the ith class and $n_{j,i}$ is the number of examples in $\mathcal{E}_j$ in the ith class, for $j = 1, 2$ and $i = 1, \ldots, c$. We define the *refinement bound*, $B_{\mathcal{P}}$, of the partition $\mathcal{P}$ by $B_{\mathcal{P}} \equiv (\max_i \{n_i + \max_{k \neq i}\ n_{1,k}\})/N$.

Theorem Let $\mathcal{E}$ be a set of examples and $\mathcal{P}$ a binary partition of $\mathcal{E}$. If $\mathcal{P}'$ is a refinement of $\mathcal{P}$, then $A_{\mathcal{P}'} \leq B_{\mathcal{P}}$.

This theorem is used by the learning system to prune the search space when searching for a predicate to split a node. During this search, the system records the best partition $\mathcal{P}$ found so far and its associated accuracy $A_{\mathcal{P}}$. When investigating a new partition $\mathcal{Q}$, the quantity $B_{\mathcal{Q}}$ is calculated. According to the theorem, if $B_{\mathcal{Q}} \leq A_{\mathcal{P}}$, then the partition $\mathcal{Q}$ and all its refinements can be safely pruned. The algorithm for enumerating predicates carefully structures the search space of predicates at a node so that the implication relation between predicates is (partially) known in order to exploit the theorem.

5.2 Enumerating Predicates

To motivate the ideas, we give an example. Consider the keys application in Section 4 again. Starting from the weakest predicate `top` (having type `Bunch -> Bool`), we show how to systematically enumerate predicates using a system of rewrites. First, here are the rewrites for this application.

```
top <= setExists1 (and4
          (projMake.top) (projNumProngs.top)
          (projLength.top) (projWidth.top))
top <= (==Abloy)
   ...
top <= (==Broad)
```

Each rewrite has the form `r <= s`, where `r` and `s` are predicates (of the same type). When `r` occurs as a redex in a predicate `p` and the rewrite `r <= s` is applied, the occurrence of `r` in `p` is replaced by `s` to produce a new predicate, denoted `p[r/s]`.

There are often many occurrences of the predicate `top` in systems of rewrites. The type of each occurrence of `top` can usually be inferred from the rewrite in which it occurs. For example, the occurrence of `top` in the head of the first rewrite has type `Bunch -> Bool`, while the first occurrence of `top` in the body has type `Make -> Bool`.

One of the most delicate aspects of predicate enumeration is how conjunction is handled. There are a number of ways of doing this. In the above rewrite system, a conjunction was set up that contains a predicate for each component of a key. Initially, these predicates are `projMake.top`, ..., `projWidth.top`. Note that each of these predicates is equivalent to `top`, so that they do not actually constrain the components. As the enumeration progresses, by systematically replacing each redex by the body of the matching rewrite, the `top`s in the conjunction are replaced by stronger predicates.

This apparently simple example hides several complications which we now explore. The first is that we must give a definition of the class of predicates under consideration. The formal definition of the class of predicates, called *standard predicates*, requires more space than we have available here but, roughly speaking, is a composition of the form

```
(f1 b1 ... bk). ... .(fn bm ... bp),
```

where each `fi` is a transformation and each `bj` is a (standard) predicate.

The next complication is the need to ensure that the children obtained from a predicate by applying rewrites to the predicate all imply the predicate. To explain how this is handled, we start with a definition.

Definition The relation $\Leftarrow$ on the class of standard predicates on the same type is defined by `p` $\Leftarrow$ `q` if `forall \x -> (p x) <== (q x)` is a logical consequence of the theory consisting of the definitions of the transformations (and associated functions).

The relation $\Leftarrow$ is a preorder (that is, a reflexive, transitive relation) on the class of standard predicates on the same type. We call $\Leftarrow$ the *implication preorder*. If `p` $\Leftarrow$ `q`, we say informally that `q` *implies* `p`.

Clearly, if applying a rewrite is going to produce a child which implies the parent, then we need to ensure that rewrites respect the implication preorder. More precisely, if `p <= q` is a rewrite, then `p` $\Leftarrow$ `q` should hold. However, respecting the implication preorder is not enough. We also have to take care to apply the rewrites at the "correct" redexes. To illustrate this, we give an example.

Example Consider the predicate `(domCard p).(<42)` and suppose there is a rewrite `p <= q` (where `q` implies `p`). Applying the rewrite, we obtain the child predicate `(domCard q).(<42)`. However, it is not true that `(domCard q).(<42)` implies `(domCard p).(<42)`.

This example shows that we must be careful to apply rewrites only at those redexes which are in a position such that the child predicate resulting from the rewrite implies the parent. In the previous example, `p` is not such a redex because of the occurrence of the `<` (although, if the `<` were replaced by `>`, then everything would be fine).

The next complication is that the search space of predicates can be a graph rather than a tree, so that it is possible to reach the same predicate by several different paths (of applications of rewrites). The classical solution to this problem, also adopted here, is to simply keep a record of all the predicates seen so far. This is done by maintaining a set, which would be implemented as a hash table, containing the predicates. Each time a child predicate is generated, the algorithm checks to see if it is in the set, adding it if it is not.

The last complication comes from the possible equivalence of syntactically distinct predicates. For example,

```
and2 (projMake.(==Rubo)) (projWidth.(==Broad))
and2 (projWidth.(==Broad)) (projMake.(==Rubo))
```

are equivalent predicates since the order of arguments for `and2` is immaterial. The problem is even worse with `and3`, which has 3! predicates in each equivalence class. We solve this by defining a total order on predicates and then ensuring the arguments to `and2`, `and3` and similar predicates, are increasingly ordered according to this order. This cuts out one of the two equivalent predicates just given.

5.3 The Algorithm

Starting from a decision tree consisting of a single leaf node, the algorithm successively chooses (according to the accuracy heuristic) the leaf node n with lowest accuracy. It then searches, using the techniques introduced earlier in this section, for a predicate p that provides a split of n, and extends the decision tree under n with two new leaf nodes, one obtained by branching on p and one by branching on the negation of p. The algorithm terminates if the chosen node n cannot be split to raise the accuracy.

As we propose using rather expressive hypothesis languages, we now discuss the important issue of the computational complexity of the learning algorithm. First, we remark that the current implementation of the algorithm is based primarily on an early implementation of the Escher programming language and could be made much more efficient. This inefficiency is manifested in the times reported below for the two larger applications. (These times are also greatly affected by the complexity of both the individuals and the predicates.) In fact, a more useful measure of computational complexity for the algorithm is the number of times it tests whether an individual satisfies a predicate. We report this number for the mutagenesis experiment below.

Using this measure (and assuming a bound on the number of nodes in the tree), the computational complexity of the algorithm is $O(|\mathcal{E}| \times p)$, where $|\mathcal{E}|$ is the number of examples and p is the maximum number of predicates generated to find a split. Finding a meaningful upper bound (smaller than the size of the complete space of predicates generated by the rewrites) for p seems difficult, if not impossible, as it depends amongst other things on how much pruning takes place. However, in practice, we do attempt to keep p as low as possible. First, the pruning strategy (safely) reduces the search space. Also it is possible to set a parameter that artificially raises the pruning threshold to further reduce the search space. Second, we have precise control over the class of predicates constructed, as we can choose the set of rewrites. This means we can make the hypothesis languages as expressive as we desire, accepting that the more expressive languages are likely to lead to longer computational times. Finally, we remark that our main interest is in the accuracy and comprehensibility of the induced theories, and that for many applications (for example, the two larger ones below) it is not important if it takes hours, or even days, to find suitable theories.

6. Two Larger Applications

We now investigate the application of the learning system to two well-known benchmark applications.

6.1 Musk Problem

The first application is the Musk problem introduced in Dietterich, Lathrop and Lozano-Pérez (1997). The main interest in this problem is that it is a multiple-instance problem which conventional learners find difficult and that it fits naturally into the knowledge representation framework that we have developed.

Briefly, the problem is to determine whether or not a molecule has a musk odour. The difficulty is the molecules generally have many different conformations and, presumably, only one conformation is responsible for the activity. This problem is similar to the keys problem considered earlier with the conformations corresponding to the keys, but the conformations themselves are rather more complicated than keys. Each conformation is a tuple of 166 floating-point numbers, where each floating-point number represents the distance in angstroms from some origin in the conformation out along a radial line to the surface of the conformation. Because there are a lot of values in each component, we performed a discretisation process on the data so that the range of values in each component was compacted to the 13 integral values in the range -6 to 6. The projections were named `proj1`, ..., `proj166`. Thus a predicate such as `proj98.(==1)` should be interpreted as meaning that along the radial line corresponding to the 98th component, the distance from the origin to the surface of the conformation lies between two particular values given in angstroms.

These considerations lead us to make the following declarations.

```
data Distance = -6 | -5 | ... | 5 | 6;
type Conformation = (Distance,..., Distance);
type Molecule = {Conformation};
```

We want to learn the function `musk :: Molecule -> Bool`. The hypothesis language contains the following transformations.

```
(==-6) :: Distance -> Bool;
   ...
(/=6) :: Distance -> Bool;
proj1 :: Conformation -> Distance;
   ...
proj166 :: Conformation -> Distance;
setExists1 :: (Conformation -> Bool) ->
                                Molecule -> Bool;
```

together with conjunction transformations on the type `Conformation`. The rewrites are as follows.

```
top <= setExist1 (and12 top ... top)
top <= proj1.(==-6)
   ...
top <= proj166.(/=6)
```

The experiments were carried out on the musk1 data set in Dietterich, Lathrop and Lozano-Pérez (1997) that contains 92 examples, of which 47 are musk and 45 are not. We ran a 10-fold cross-validation (taking about 12 hours) and found the average accuracy was 83.46% with a standard deviation of 9.51%. This accuracy is similar to that obtained by another general purpose decision-tree learner (Blockeel & De Raedt, 1998), but lower than the 92.4% accuracy obtained by a learning system specially developed for this problem and described in Dietterich, Lathrop and Lozano-Pérez (1997). The learner found the following definition for the function `musk` on the full data set of 92 examples.

```
musk m =
  if setExists1 (and8
      (proj98.(==1)) (proj84.(/=-3))
      (proj65.(/=3)) (proj65.(/=2))
      (proj13.(/=-6)) (proj10.(/=-4))
      (proj8.(/= 2)) (proj3.(/= 3))) m
  then True
  else False;
```

6.2 Mutagenesis Problem

The next application involves learning a theory to predict whether a chemical molecule is mutagenic or not (King et al., 1996). As before, we use an (undirected) graph to model a molecule. The declarations and hypothesis language are similar to the graph application in Section 4. The examples used were the set of 188 regression-friendly molecules. In the first experiment, we used the atom/bond information, and I_1, I_a and ϵ_{LUMO}. On a 10-fold cross-validation (taking about 5.5 hours), the average accuracy was 87.3% (similar to King et al (1996), and Blockeel and De Raedt (1998)) with a standard deviation of 6.10%. On the entire data set, the definition produced was that "a molecule is mutagenic iff I_1 is true or $\epsilon_{LUMO} < -2.368$".

In the second experiment, we used only the atom/bond information. We ran a 10-fold cross-validation (taking about 31 hours) and found the average accuracy was 79.75% (compared to 79% in Blockeel and De Raedt (1998)) with a standard deviation of 8.48%. On average, 7026 predicates were tested against all (170) examples, of which 86% caused pruning. The learner found the following definition on the full data set.

```
mutagenic m =
 if vertices.(domCard (and2
         (vertex.projCharge.(>=0.29))
         (vertex.projAtomType.(==1)))).(>0)
 then if vertices.(domCard (and2
        (vertex.projCharge.(>=-0.424))
        (vertex.projAtomType.(==34)))).(>1)
     then True
     else False
 else if vertices.(domCard
         (vertex.projAtomType.(==50))).(>0)
     then False
     else if vertices.(domCard
         (vertex.projAtomType.(==34))).(>1)
       then False
       else if vertices.(domCard
            (vertex.projAtomType.(==35))).(>0)
                  then False
                  else True;
```

"If a molecule has an atom with partial charge ≥ 0.29 and atom type 1, then if it has at least two atoms with partial charge ≥ -0.424 and atom type 34, then it is mutagenic, else it is not mutagenic; else if it has an atom with atom type 50, then it is not mutagenic; else if it has at least two atoms with atom type 34, then it is not mutagenic; else if it has an atom with atom type 35, then not mutagenic, else it is mutagenic".

7. Discussion

Inductive logic programming (ILP) (Muggleton & De Raedt, 1994) addresses exactly the same class of problems as we do in this paper, but by rather different methods. Most researchers in ILP use an (untyped) first-order logic for knowledge representation and Prolog as the implementation mechanism. The main difference between the approach of ILP and that adopted here is in the handling of knowledge representation. ILP employs an untyped, first-order logic; we use a typed higher-order logic. (But note that some ILP researchers, recognising the value of types, have employed limited, *ad hoc* type systems to reduce the search space.) ILP represents individuals by a database of facts pertinent to that individual; we employ a (closed) term.

We now turn to specific related work. One paper which advocates the usefulness of sets in machine learning is Cohen (1996). This paper studies set-valued features in decision-tree learning. The context there is rather more restricted than this paper as only sets of strings are allowed. However, the approach and motivation are similar, as are the predicates generated for the set-valued features. Furthermore, Cohen (1996) makes a strong case for the practicality of using set-valued features. Another paper by Cohen and Hirsh (1994) studies learnability in description logics that have certain similarities to higher-order logics. The existing learning system most closely related to ours is the Tilde system, described in Blockeel and De Raedt (1998). The motivation of extending attribute-value decision-tree learning to richer representation languages also drives the development of Tilde and, in general terms, it is similar in architecture and can handle similar problems to our learner, but there are several significant differences. In particular, Tilde is based on untyped first-order logic and it can spread conditions and variables across several nodes of the tree. Finally, we note that a number of authors have previously suggested using conjunctions of conditions to split nodes.

8. Conclusion

We have proposed using typed, higher-order logic for representing individuals with complex internal structure and for hypothesis languages. An attractive feature of this approach is that it allows learning algorithms to have precise control over the hypothesis spaces generated. Further work includes improving the efficiency of the implementation, studying some large-scale applications, and extending the theoretical foundations.

Acknowledgements

We thank Kee Siong Ng who found several improvements to the learning algorithm and performed many useful experiments, and also the members of the Machine Learning Research Group at Bristol University. This work was partially supported by EPSRC Grant GR/L21884 and the Australian National University.

References

Blockeel, H., & De Raedt, L. (1998). Top-down induction of first-order logical decision trees. *Artificial Intelligence*, *101*, 285–297.

Cohen, W. (1996). Learning trees and rules with set-valued features. *Proceedings of the Thirteenth National Conference on Artificial Intelligence* (pp. 709–716). Menlo Park, CA: AAAI Press.

Cohen, W., & Hirsh, H. (1994). Learning the CLASSIC description logic: Theoretical and experimental results. *Proceedings of Fourth International Conference on Principles of Knowledge Representation and Reasoning* (pp. 121–133). Morgan Kaufmann.

Dietterich, T., Lathrop, R., & Lozano-Pérez, T. (1997). Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, *89*, 31–71.

King, R., Muggleton, S., Srinivasan, S., & Sternberg, M. (1996). Structure-activity relationships derived by machine learning: The use of atoms and their bond connectivities to predict mutagenicity in inductive logic programming. *Proceedings of the National Academy of Sciences*, *93*, 438–442.

Lloyd, J. (1999). Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming, Volume 1999, Article 3.*

Muggleton, S., & De Raedt, L. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, *19/20*, 629–679.