A Knowledge Representation Framework for Inductive Learning

A.F. Bowers[†], C. Giraud-Carrier[‡] and J.W. Lloyd[§]

[†]STMicroelectronics Ltd, Bristol, U.K.

[‡]ELCA Informatique SA, Lausanne, Switzerland

[§]Computer Sciences Laboratory, Research School of Information Sciences and Engineering, Australian National University

Antony.Bowers@st.com, cgc@elca.ch, jwl@csl.anu.edu.au

Abstract

This paper provides a foundation for knowledge representation for inductive learning based on the use of higher-order logic. 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) gives a systematic way of constructing predicates for use in induced definitions; (iii) provides a general framework for feature extraction, construction, and selection; (iv) widens the applicability of decision-tree algorithms beyond the usual attribute-value setting to the classification of individuals with complex structure; and (v) shows how to induce definitions that are comprehensible and have predictive power. The paper contains ten illustrative applications involving a variety of types to which a decision-tree learning system is applied. The effectiveness of the approach is further demonstrated by applying the learning system to two larger benchmark applications.

1 Introduction

Inductive learning focuses on techniques for supervised learning from examples. Traditionally, inductive learners have used the attribute-value language to represent training examples and induced hypotheses. The relative simplicity of this attribute-value language representation allows the implementation of efficient learning systems. However, it also makes it difficult to apply machine learning to domains where data is rich in structure. In addition, the underlying structural information may actually prove essential in inducing insightful concepts. As Quinlan [Qui96] puts it: “Data may concern objects or observations with arbitrarily complex structure that cannot be captured by the values of a predetermined set of attributes”. For example, in a chemical domain, the individuals[1] under consideration may be molecules, where molecules are complex arrangements of atoms each with their own set of characteristics (e.g., type, charge, etc.).

In such cases of complex structure in the data, the traditional approach requires that each individual’s structure be captured by a tuple of constants. One can easily argue that, from a knowledge representation point of view, it would be far more desirable to capture directly the structures in the data with corresponding structures in its representation. For example, it can make good sense to represent a molecule, which is a collection of connected atoms, as a graph; or a bunch of keys, which is a collection of keys on a key ring, as a set. Then, if we can design learning algorithms capable of manipulating such representations directly, it would be possible to apply machine learning more directly and naturally to domains rich in structure.

It is instructive to examine in more detail the ideas of the previous paragraph. First, there is a space of individuals (the *individual space*) for which there is some learning task. In individual

[1]The terminology ‘instance’ is often used elsewhere instead of ‘individual’; we employ ‘individual’ throughout the paper.

space, the representation of each individual should capture in an appropriate way the structure of that individual. For example, a molecule may be represented as a graph in which the atoms are vertices and the edges are bonds. This requirement to suitably capture the structure of each individual implies that the representation language for the individual space must include data types such as tuples, sets, multisets, lists, graphs, and so on. This is illustrated in Figure 1 in which a particular individual in individual space is shown. This individual is represented by a set of tuples in which the first and second arguments of the tuple are constants and the third argument is a list. Individuals as complicated as this are not uncommon – there are even more complicated ones later in the paper. In the formalism introduced in this paper, individual space is a space of certain closed, typed, higher-order terms that can be made into a metric space in a natural way [Llo01a, Llo01b]. This formalism provides a rich representation language for individuals and, in particular, provides tuples, sets, multisets, lists, and (somewhat indirectly) graphs. A common alternative to higher-order logic for representation is first-order logic for which an individual is normally named by a constant and then represented by a relational database of facts about the constant. This is the usual approach of the subfield of inductive logic programming (ILP). (This matter is discussed further in Section 9.)

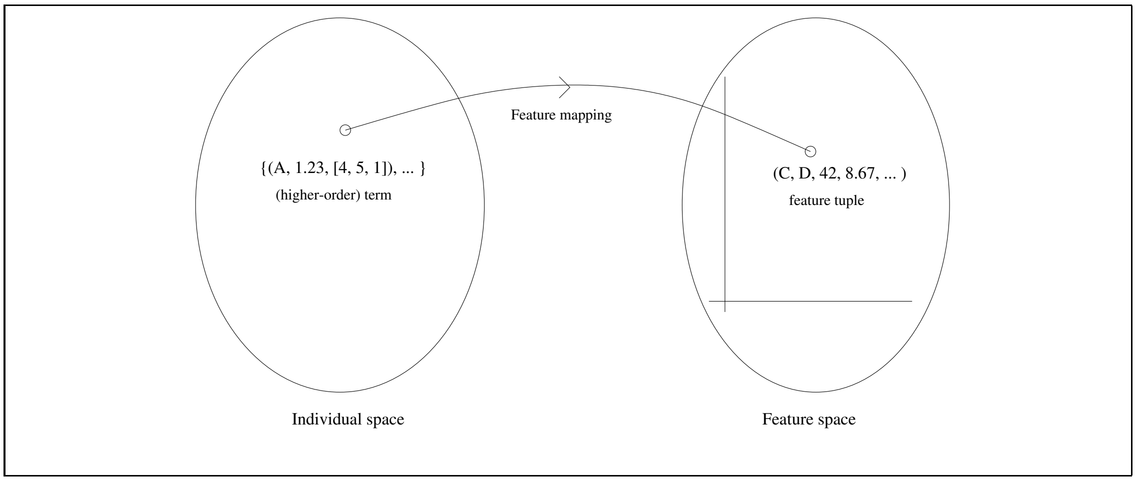

Figure 1: Feature mapping from individual space to feature space

Second, there is a space of feature tuples (the *feature space*) where each argument in a feature tuple is a number or a constant. (Feature tuples are usually called feature vectors – the terminology of tuple is adopted here since it is commonly used in logic.) The representation language of feature space is the attribute-value language. Thus feature space is a product space of the form $R_1 \times \ldots \times R_n$, for some $n \geq 1$, where each R_i is either the floating-point numbers, the integers, or some finite set of constants (with or without an order). The axes in the codomain in Figure 1 are meant to indicate a product space. In Figure 1, a feature tuple with constants in the first two arguments, an integer in the third argument, a floating-point number in the fourth argument, and so on, is indicated. Feature tuples are used to represent individuals for input purposes for the vast majority of learning systems. The major exceptions are ILP systems that use first-order logic, as indicated in the previous paragraph.

Third, there is a mapping (the *feature mapping*) from individual space to feature space whereby an individual is uniquely associated with a feature tuple. Intuitively, the structure of a term representing an individual in individual space is "flattened" by the feature mapping into a tuple in feature space, where each feature is some numerical or constant-valued property of the individual. Often, both the individual space and this mapping are only implicit and the learning task begins with a feature-tuple representation of the individuals of interest. This may happen if the learning task starts out with some given, previously generated attribute-value data about the individuals. However, there may also be the opportunity of starting the representation process with individual space, in which case considerable extra flexibility may result. In this case, there are two possible ways to proceed. The usual approach is to define a mapping into feature space and then apply conventional learning techniques to the feature tuples representing the individuals. The second approach is to avoid the *fixed* set of features defining feature space and deal directly with all the possible features generated directly on individuals. This approach requires the generalisation of conventional learning algorithms, which usually apply to feature tuples, to the more general situation of applying directly to individuals (in our case, certain higher-order terms). We illustrate this second approach by showing how a conventional decision-tree learner can be generalised to deal with the representation requirements of individual space.

Clearly, there are various trade-offs for the two approaches. By mapping into feature space, the first approach has the advantage that conventional learning techniques are immediately applicable and also that these algorithms are likely to be more efficient since they operate on the simpler feature-tuple representation of individuals. But there is also a disadvantage in that the choice of features, and generally there are many potential features, needs to be a good one, otherwise the learning algorithm will have difficulty. The second approach addresses this difficulty since the learning algorithm has direct access to a much larger collection of possible features while it is searching for a suitable target function. However, this flexibility brings with it the disadvantage that the corresponding search space is likely to be much larger than for the first approach and therefore the generalised learning algorithm is likely to be less efficient than one operating on tuples of (previously selected) features. In summary, one can characterise the first approach as a *static* (the features are selected first and then the learning algorithm is run on the feature-tuple representation), while the second approach is *dynamic* (the learning algorithm is run on the term representation and features are constructed as needed during learning).

This paper concentrates on the dynamic approach working directly with the term representation of individuals, at the same time recognising that the extra flexibility of this approach may incur the cost of longer execution times for the learning algorithms, since there are applications, such as molecular biology, where the emphasis is on gaining insight from the data. Hence, the quality, as measured by both accuracy and comprehensibility, of the induced knowledge is more important than the execution time of the learning algorithm. Furthermore, the extra complexity in the search merely reflects the *intrinsic* complexity of the application itself – if the individuals have complex structure, then inevitably there will be a large number of possible features that could be used to classify them.

While not explored explicitly here, the paper also provides a general framework for the static approach of feature extraction, construction and selection [LM98a, LM98b] in that it develops a rich representation language for individuals together with a systematic method for constructing potentially useful features for the individuals.

In summary, what follows is a new foundation for knowledge representation for inductive learning that naturally extends the attribute-value framework to account for applications where the individuals to be classified have complex structure. This foundation is in the tradition of, and generalises, other logic-based approaches to inductive learning such as [Plo70], [Sam81], [Sha83], [Mic83], and [MD94]. We use a typed, higher-order logic as the knowledge representation formalism. This logic provides highly expressive hypothesis languages, which support a variety of important 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. Furthermore, these definitions can be translated straightforwardly to a structured English format that can be understood by users without knowledge of (higher-order) logic. An account of the theoretical foundations of the use of higher-order logic in inductive learning is contained in [Llo01b] and further discussion on knowledge representation issues, especially those concerning types, is contained in [FGCL98]. An outline of the results of this paper was given in [BGCL00]. We illustrate the ideas of the paper on a variety of problems. We also describe a decision-tree learning system based on the accuracy heuristic. The effectiveness of the approach is demonstrated by applying the learning system to two benchmark applications: Musk, a typical multiple-instance problem, and Mutagenesis, a problem of practical interest to chemists and biologists.

An overview of the paper is as follows. Section 2 defines the inductive learning problem formally. Section 3 describes how individuals are represented in higher-order logic. Section 4 contains a discussion of particular functions called transformations that are composed to create predicates. Section 5 contains ten illustrative applications, several of which are taken from the literature, and reformulated and solved in our setting. Section 6 describes predicate construction. Section 7 contains a general discussion of accuracy-driven, decision-tree learning algorithms and also describes our decision-tree learning system. Section 8 describes the application of this learner to the Musk and Mutagenesis problems. Section 9 contains a discussion of what has been achieved in this paper, a comparison with some related work, especially in inductive logic programming, and also a comparison of our learning system with some other systems. Finally, Section 10 contains some conclusions. An appendix contains the definitions in higher-order logic of basic terms, which are used for knowledge representation, and regular predicates, which are the output of predicate construction.

2 The Inductive Learning Problem

This section contains a general formulation of the inductive learning problem.

Inductive learning covers a spectrum of applications from program synthesis to classification. While the formulation below is sufficiently general to cover all applications in this spectrum, we are primarily interested in classification problems in this paper. One is given some domain of individuals for which a general classification is required. The classification will be given by a function from this domain to some small finite set corresponding to the classes. Examples are available that provide the class of some typical individuals. In addition, some background knowledge relevant to the inductive task at hand may be available. From this, the general classification of the individuals has to be induced.

The inductive learning problem can be summarised formally as follows. We assume that there is an underlying alphabet which is dependent on the problem and extensive enough to contain all symbols appearing in the components below.
Given:

1. *Signature of the target function.*
   The target function $f : \sigma \to \tau$ is the function whose definition has to be learned from examples. Here σ is the type of the domain and τ is the type of the codomain of f.

2. *Examples.*
   These have the form $(f\ s) = t$, where s and t are closed terms of type σ and τ respectively.

3. *Hypothesis language.*
   This is the language used to construct the definition of the target function.

4. *Background theory.*
   The background theory consists of the definitions of the functions which appear in (the alphabet of) the hypothesis language.

5. *Cover relation.*
   We say that a definition for the target function f *covers* an example $(f\ s) = t$ if the term $(f\ s) = t$ is a logical consequence of the theory consisting of the induced definition for f and the background theory.

Find:

A *definition* for the target function that is in the hypothesis language and which

1. covers the examples, and

2. generalises (that is, accurately classifies new individuals).

The above formulation is a natural extension of those given by Mitchell [Mit82] and Michalski [Mic83]. We now make some general remarks about the various components of the inductive learning problem.

In the context of classification, τ is the type of a simple codomain consisting of just a few elements corresponding to the classes. Often τ is the booleans. On the other hand, individuals of type σ can have arbitrarily complex structure. In the case of attribute-value learning, individuals are represented by tuples of constants. We generalise here to individuals with more structure by allowing more complex, possibly higher-order, types to be used in the representation. Note that the reformulation of attribute-value learning presented here is, in a sense, "isomorphic" to the traditional one that 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 *generalises* directly to the representation of individuals with complex structure. Indeed, we concentrate here on applications which would be difficult or impossible to represent using just the syntax of tuples of constants.

The hypothesis language is determined by an alphabet of symbols and the syntactic form of expressions that may appear in the language. In attribute-value problems, symbols that are likely to appear include *if_then_else*, $=$, $\geq$, $>$, $\leq$, $<$, $\wedge$, $\vee$, $\neg$, and the projections of tuples onto their components. (The projections extract the value of a particular attribute.) When more complicated types are involved, the alphabet is richer, as we shall see below. The syntactic form of the expressions that may appear in the hypothesis language is determined by the language bias [Mit97, p.64] that provides prior assumptions about the form of the definition to be learned. Part of the language bias may be implicit in the architecture of the learning system, but it is also common to allow the language bias to be specified explicitly by pragma, which are hints/directives to the system that constrain the hypothesis language. For example, the user could specify the patterns (that is, particular syntactic forms) which can appear in the generated hypotheses. In this paper, the hypothesis language is specified by the user by means of a system of rewrites. The other kind of bias is search bias that restricts how much of the search space corresponding to the hypothesis language will be searched. Typically, search bias is introduced by the use of heuristics in the search process. Here the heuristic used for the decision-tree algorithm is the accuracy heuristic.

Next we turn to the background theory which consists of the definitions of the functions that appear in the alphabet of the hypothesis language. Note that there are various kinds of background knowledge. For each complex type, one needs functions for creating predicates appropriate for each type. For example, the functions *card* on sets and *mcard* on multisets discussed below reside in the background theory. The remainder of the background theory consists of the definitions of auxiliary functions that are neither useful outside the context of a particular learning task nor to be easily found by the learner itself. Generally, these definitions have to be provided by the user.

Depending on the application, the induced definition may cover some or all of the examples. In a program synthesis application, it is likely that the induced definition will be required to cover every example. However, most applications have noise in the data and for these the induced definition may only be required to cover some percentage of the examples. Primarily, we want to induce a definition of the target function which is *comprehensible*, that is, gives insight on the nature of the data, and has *predictive power*, that is, accurately predicts the class of new individuals.

3 Representation of Individuals

This section contains a discussion of the use of higher-order logic to represent individuals.

We employ a higher-order logic as the knowledge representation formalism and 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 syntax is typically needed to represent individuals? A strong clue to this is given by the examples in Section 5. Typically, one needs the following:

- Integers, natural numbers, floats, characters, strings, and booleans.
- Data constructors.
- Tuples.
- Sets.
- Multisets.
- Lists.
- Trees.
- Graphs.

The first group, called the *base types*, are the basic building blocks of terms representing individuals. The type of integers is denoted by Int, the type of natural numbers by Nat, the type of floats by $Float$, the type of characters by $Char$, the type of strings by $String$, and the type of booleans by Ω.

It is convenient to be ambivalent in the treatment of strings. If strings are regarded as atomic types, with no internal structure, then they belong in the collection of base types. However, for some applications, the internal structure of the strings is important and for these it is best to regard a string as a list of characters. Commonly, the type $String$ is a synonym for $List\ Char$, in which case these views coincide. However, for machine learning purposes, we treat $String$ and $List\ Char$ differently: $String$ is regarded as a base type and the predicates generated are those for a base type, while a term of type $List\ Char$ is regarded as a list with predicates corresponding to lists.

Also needed are data constructors for user-defined types. For example, see the data constructors *Sunny*, *Overcast*, and *Rain* for the type *Outlook* in Subsection 5.1 below. These data constructors have arity 0 and are usually called *constants*. Data constructors for more complicated types are also needed (e.g., the cons function (:) for lists).

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 here. 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 that best models the application being studied. While much can be done with tuples of constants, there are definite limits to what can be achieved with such simple machinery. For example, in an application we study below, a bunch of keys is modelled by a set, where each item in the set models a key. Having a set at the top level like this turns up naturally in multiple-instance problems [DLLP97]. Here is an example naturally involving graphs.

Example. As an example of knowledge representation, consider the problem of modelling a chemical molecule. The first issue is to choose a suitable type to represent a molecule. We use an undirected graph to model a molecule – an atom is a vertex in the graph and a bond is an edge. Having made this choice, suitable types are then set up for the atoms and bonds. For this, we first introduce the type $Element$, which is the type of the (relevant) chemical elements.

$$Br, C, Cl, F, H, I, N, O, S : Element.$$

This declaration declares the constants *Br* (bromine), *C* (carbon), and so on, to be constants of type *Element*. We also make the following type synonyms.

$$\begin{aligned}
&AtomType = Nat\\
&Charge = Float\\
&Atom = Element \times AtomType \times Charge\\
&Bond = Nat.
\end{aligned}$$

The expression '$AtomType = Nat$', for example, simply states that the type *AtomType* is nothing other than *Nat*. The use of type synonyms greatly improves the readability of declarations.

The type of an (undirected) graph is $Graph\ \nu\ \varepsilon$, where ν is the type of information associated with the vertices and ε is the type of information associated with the edges. *Graph* is defined as follows.

$$\begin{aligned}
&Label = Nat\\
&Graph\ \nu\ \varepsilon = \{Label \times \nu\} \times \{(Label \to Nat) \times \varepsilon\}.
\end{aligned}$$

Here the multisets of type $Label \to Nat$ are intended to all have cardinality 2, that is, they are intended to be regarded as *unordered* pairs. (A multiset is a function mapping into the natural numbers for which the value of the function on some element is its multiplicity, that is, the number of times it occurs in the multiset.) Note that this definition corresponds closely to the mathematical definition of a graph: each vertex is labelled by a unique natural number and each edge is uniquely labelled by the unordered pair of labels of the vertices that it connects.

We now declare the type of a molecule to be an (undirected) graph whose vertices have type *Atom* and whose edges have type *Bond*. This leads to the following type synonym.

$$Molecule = Graph\ Atom\ Bond.$$

Here is an example molecule, called d_1, from the mutagenesis dataset available at [Yor]. The notation $\langle s, t\rangle$ is used as a shorthand for the multiset that takes the value 1 on each of s and t, and is 0 elsewhere. Thus $\langle s, t\rangle$ is essentially an unordered pair.

$$\begin{aligned}
&(\{(1, (C, 22, -0.117)), (2, (C, 22, -0.117)), (3, (C, 22, -0.117)),\\
&\quad (4, (C, 195, -0.087)), (5, (C, 195, 0.013)), (6, (C, 22, -0.117)),\\
&\quad (7, (H, 3, 0.142)), (8, (H, 3, 0.143)), (9, (H, 3, 0.142)),\\
&\quad (10, (H, 3, 0.142)), (11, (C, 27, -0.087)), (12, (C, 27, 0.013)),\\
&\quad (13, (C, 22, -0.117)), (14, (C, 22, -0.117)), (15, (H, 3, 0.143)),\\
&\quad (16, (H, 3, 0.143)), (17, (C, 22, -0.117)), (18, (C, 22, -0.117)),\\
&\quad (19, (C, 22, -0.117)), (20, (C, 22, -0.117)), (21, (H, 3, 0.142)),\\
&\quad (22, (H, 3, 0.143)), (23, (H, 3, 0.142)), (24, (N, 38, 0.812)),\\
&\quad (25, (O, 40, -0.388)), (26, (O, 40, -0.388))\},\\
&\ \{(\langle 1, 2\rangle, 7), (\langle 1, 6\rangle, 7), (\langle 1, 7\rangle, 1), (\langle 2, 3\rangle, 7), (\langle 2, 8\rangle, 1),\\
&\quad (\langle 3, 4\rangle, 7), (\langle 3, 9\rangle, 1), (\langle 4, 5\rangle, 7), (\langle 4, 11\rangle, 7), (\langle 5, 6\rangle, 7),\\
&\quad (\langle 5, 14\rangle, 7), (\langle 6, 10\rangle, 1), (\langle 11, 12\rangle, 7), (\langle 11, 17\rangle, 7),\\
&\quad (\langle 12, 13\rangle, 7), (\langle 12, 20\rangle, 7), (\langle 13, 14\rangle, 7), (\langle 13, 15\rangle, 1),\\
&\quad (\langle 14, 16\rangle, 1), (\langle 17, 18\rangle, 7), (\langle 17, 21\rangle, 1), (\langle 18, 19\rangle, 7),\\
&\quad (\langle 18, 22\rangle, 1), (\langle 19, 20\rangle, 7), (\langle 19, 24\rangle, 1), (\langle 20, 23\rangle, 1),\\
&\quad (\langle 24, 25\rangle, 2), (\langle 24, 26\rangle, 2)\}).
\end{aligned}$$

The formal basis for the representation of individuals is provided by the concept of a *basic term*. An outline of the theoretical development of basic terms is given in the Appendix, and much more detail is in [Llo01b]. Essentially, one first defines the concept of a term in higher-order logic. Next one defines basic terms via an inductive definition that has three parts: the first part gives those basic terms that have a data constructor at the top level, the second part gives certain abstractions that include (finite) sets and multisets, and the third part gives tuples. Care is taken to order subterms of abstractions to ensure uniqueness of the representation.

4 Transformations

We now discuss a particular class of functions, called transformations, and how they can be composed to create predicates. Further detail about predicate construction is given in Section 6 and the Appendix. The theoretical foundations of this topic are given in [Llo01b].

A predicate is a function whose codomain is the booleans. Thus predicates are the logical equivalent of (boolean-valued) features in traditional machine learning, where features capture salient properties that individuals may or may not possess (e.g., whether a molecule contains a carbon atom or not). In what follows, we prefer to use the term 'predicate' instead of 'feature'.

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 possibly large space of predicates. This search space is made up by composing more basic functions in all possible ways. So we need to say something about what these more basic functions are and how they are composed into predicates.

First, we deal with composition. Composition is handled by the (reverse) composition function.

$$\cdot : (\rho \to \sigma) \to (\sigma \to \tau) \to (\rho \to \tau)$$
$$((f \cdot g)\ x) = (g\ (f\ x)).$$

This function definition is typical: it consists of a signature[2] for the function, followed by one or more equations satisfied by the function that are used for computation. Here, and this is also done for some other functions without comment, the function $\cdot$ is written infix. The signature for $\cdot$ is given in so-called "curried" form with arrows between the types of the two arguments. This form is convenient mainly because it allows the possibility of *partial application*, that is, the application of a function to some but not all of its arguments. For example, one can form $(\cdot\ f)$ which is the application of $\cdot$ to its first argument f that produces a function of type $(\sigma \to \tau) \to (\rho \to \tau)$. Note that this definition of composition implies that the component functions are applied in left to right order.

Predicates are built up by composing transformations, which are defined as follows.

Definition. A *transformation* f is a function having a signature of the form

$$f : (\rho_1 \to \Omega) \to \cdots \to (\rho_k \to \Omega) \to \mu \to \sigma,$$

where any parameters in $\rho_1, \ldots, \rho_k$ and σ appear in μ, and $k \geq 0$. The type μ is distinguished and is called the *source* of the transformation, while the type σ is called the *target* of the transformation. The number k is called the *rank* of the transformation.

[2]More formally, the signature for $\cdot$ is $(a \to b) \to (b \to c) \to (a \to c)$, where a, b, and c are parameters (that is, type variables) that can be instantiated with any types. The composition function is thus *polymorphic* in the sense that it can be used to compose functions of any type provided that the type of the codomain of the first function is the same as the type of the domain of the second function. Many later functions are also polymorphic, but we suppress explicit mention of this by replacing the parameters by types (usually denoted by μ, ρ, and so on) that are assumed appropriate to the particular application.

The intuitive idea behind this definition is that, given predicates $p_i : \rho_i \to \Omega$, for $i = 1, \ldots, k$, $f\ p_1 \ldots p_k$ is a function which takes individuals of type μ to individuals of type σ. By composing (generally) several such functions, the last of which is a predicate, a predicate on individuals of the desired type is obtained. A special case is when $k = 0$, in which case the transformation is itself such a suitable function. These ideas will be illustrated below by various examples.

We now introduce some transformations that will be used often in the following. The first of these transformations is defined on sets, so we need to say first how sets are introduced. In higher-order logic, a set is identified with a predicate, its characteristic function. Thus a set has type $\mu \to \Omega$, for some type μ. However, in practice, even though sets and predicates have been identified, for a term of type $\mu \to \Omega$, it is sometimes convenient to make an informal distinction depending upon whether one is thinking of it as a 'set of elements' or as a 'condition'. In the former case, the synonym $\{\mu\}$ for $\mu \to \Omega$ is used to indicate this. To emphasise, there is no mathematical distinction between $\{\mu\}$ and $\mu \to \Omega$, but there is a change in the intuitive role the corresponding term is understood to be playing. Similarly, the notation '$x \in t$' is used if one is thinking of t as a 'set of elements' and the notation '$(t\ x)$' is used if one is thinking of t as a 'condition'. This distinction will be convenient in the following discussion.

Now consider the transformation

$$
\begin{aligned}
&\mathit{setExists}_1 : (\mu \to \Omega) \to \{\mu\} \to \Omega \\
&\mathit{setExists}_1\ b\ t = \exists x.(b\ x) \wedge (x \in t).
\end{aligned}
$$

Then, if b is a predicate on the type μ, $(\mathit{setExists}_1\ b)$ is a function of type $\{\mu\} \to \Omega$, that is, a predicate on sets whose elements have type μ. This predicate is true of a set iff it has an element which satisfies the predicate b. Consider the transformation

$$
\begin{aligned}
&\mathit{setExists}_2 : (\mu \to \Omega) \to (\mu \to \Omega) \to \{\mu\} \to \Omega \\
&\mathit{setExists}_2\ b\ c\ t = \exists x.\exists y.(b\ x) \wedge (c\ y) \wedge (x \in t) \wedge (y \in t) \wedge (x \neq y).
\end{aligned}
$$

The predicate $(\mathit{setExists}_2\ b\ c)$ checks whether a set has distinct elements x and y such that x satisfies b and y satisfies c. More generally, for $n \geq 1$, one can define $\mathit{setExists}_n$.

$$
\begin{aligned}
&\mathit{setExists}_n : (\mu \to \Omega) \to \cdots \to (\mu \to \Omega) \to \{\mu\} \to \Omega \\
&\mathit{setExists}_n\ p_1 \ldots p_n\ t = \exists x_1. \cdots \exists x_n.\ (p_1\ x_1) \wedge \cdots \wedge (p_n\ x_n) \wedge \\
&\qquad\qquad (x_1 \in t) \wedge \cdots \wedge (x_n \in t) \wedge (x_1 \neq x_2) \wedge \cdots \wedge (x_{n-1} \neq x_n).
\end{aligned}
$$

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

$$
\begin{aligned}
&\wedge_n : (\mu \to \Omega) \to \cdots \to (\mu \to \Omega) \to \mu \to \Omega \\
&\wedge_n\ p_1 \ldots p_n = \lambda x.((p_1\ x) \wedge \cdots \wedge (p_n\ x)),
\end{aligned}
$$

where $n \geq 2$, that provides a "conjunction" with n conjuncts.

Corresponding to disjunction, we have the transformation

$$
\begin{aligned}
&\vee_n : (\mu \to \Omega) \to \cdots \to (\mu \to \Omega) \to \mu \to \Omega \\
&\vee_n\ p_1 \ldots p_n = \lambda x.((p_1\ x) \vee \cdots \vee (p_n\ x)),
\end{aligned}
$$

where $n \geq 2$.

Negation is handled by the usual connective $\neg$ which is a transformation of type $\Omega \to \Omega$. Thus $p \cdot \neg$ is the negation of the predicate p.

Going back to the basic types, if N is an integer, then

$$(= N) : \mathit{Int} \to \Omega$$

is a transformation, in fact a predicate, defined by

$$(= N)\ m = m = N.$$

(Here we are exploiting the curried syntax: the signature for $=$ is $a \to a \to \Omega$, so that $(= N)$ is the application of $=$ to just its first argument.) Similarly, one can define transformations $(< N)$ corresponding to $<$, $(> N)$ corresponding to $>$, $(\neq N)$ corresponding to $\neq$, and so on.

More generally, if μ is a type and C a constant of type μ, then we can make a predicate $(= C) : \mu \to \Omega$ by defining $(= C)\ x = x = C$. Similarly, the predicate $(\neq C) : \mu \to \Omega$ is defined by $(\neq C)\ x = x \neq C$. Note that $\neg$ is the same as $(= \bot)$. ($\bot$ is the constant with intended meaning *False*.)

For any type μ, there is a transformation $top : \mu \to \Omega$ defined by $top\ x = \top$. ($\top$ is the constant with intended meaning *True*.) Thus *top* is the predicate which always returns $\top$. Similarly, $bottom : \mu \to \Omega$ is the predicate which always returns $\bot$. It is taken for granted that each hypothesis language defined in the following includes the transformation *top*, for each relevant type.

There is an if-then-else transformation defined as follows.

$$ite : (\mu \to \Omega) \to (\mu \to \Omega) \to (\mu \to \Omega) \to \mu \to \Omega$$
$$ite\ p\ q\ r = \lambda x.if\ (p\ x)\ then\ (q\ x)\ else\ (r\ x).$$

As an illustration, one could define the predicate $ite\ (null \cdot \neg)\ (head \cdot (= 42))\ bottom$, where *null* checks whether a list is empty or not and *head* extracts the head of a non-empty list. This predicate returns $\top$ iff its argument is a non-empty list with 42 as its first item.

Before discussing each of the types in more detail, we make some remarks about the contrast between generic and domain-specific transformations. In this paper, we concentrate on generic transformations, that is, given some type we provide a set of transformations that are likely to be generally useful in applications involving that type. We do not claim to have given *all* generic transformations for each type, just useful collections so that any particular application will probably only need a subset of the ones we provide. However, it is common in applications to also have need of domain-specific transformations which capture some specialised concept that will be useful in the application. For example, in the Mutagenesis problem quite a number of domain-specific predicates have been studied, including such things as the possible existence of 2-dimensional substructures like benzene rings and methyl groups [KMSS96, SKM99]. Such predicates are not relevant in general graphs, but they can be important if the graphs are a representation of molecules. Technically, such domain-specific transformations are handled in exactly the same way as generic transformations are handled in this paper. Domain-specific transformations are added to the hypothesis language. Their definitions become part of the background theory and can be used in the construction of predicates in the same way as generic transformations.

We now discuss each of the types more systematically, giving various transformations and predicates defined on them.

4.1 Tuples

(Ordered) tuples are, in effect, already widely used in the case of attribute-value applications. The transformations admitted for a tuple of type $\mu_1 \times \cdots \times \mu_n$ are the projections

$$proj_i : \mu_1 \times \cdots \times \mu_n \to \mu_i$$
$$proj_i\ (t_1, \ldots, t_n) = t_i,$$

for $i = 1, \ldots, n$. Thus $proj_i$ projects onto the ith component of a tuple. Each $proj_i$ is a transformation of rank 0.

As an example of predicate construction, suppose C is a constant of type μ_1. Then, by composition, we can construct the predicate $proj_1 \cdot (= C)$ on individuals of type $\mu_1 \times \cdots \times \mu_n$. This predicate is true of an individual of this type iff its first component is C.

4.2 Sets

We make a great deal of use of sets in knowledge representation because the set type is a fundamental type and many applications (or parts of applications) are best modelled as sets. For example, we show below how sets are used to model multiple-instance learning applications.

We have already introduced the transformations $setExists_n$, for $n \geq 1$, on sets. Here is another useful transformation. For sets whose elements have type μ, consider the transformation

$$domCard : (\mu \to \Omega) \to \{\mu\} \to Nat$$
$$domCard\ b\ t = card\ \{x \mid (b\ x) \wedge x \in t\},$$

where *card* computes the cardinality of a (finite) set. Thus, for each predicate b, the function

$$(domCard\ b) : \{\mu\} \to Nat$$

computes the cardinality of the subset of elements of the argument that satisfy the predicate b.

As an example of predicate construction, one can then obtain a predicate by composing $(domCard\ b)$ with the predicate $(> N)$, for some $N \geq 0$. This gives the predicate $(domCard\ b) \cdot (> N)$, which is true of a set iff it contains more than N elements satisfying the predicate b.

Another transformation for sets is

$$setAll : (\mu \to \Omega) \to \{\mu\} \to \Omega$$
$$setAll\ b\ t = \forall x.(x \in t \longrightarrow (b\ x)).$$

Thus $(setAll\ b)$ is true of a set iff every element in the set satisfies the predicate b.

4.3 Multisets

Multisets (also known as bags) are a useful generalisation of sets. A straightforward approach to multisets is to regard a multiset as a function of type $\mu \to Nat$, where μ is the type of elements in the multiset and the value of the multiset on some item is its multiplicity, that is, the number of times it occurs in the multiset. (In fact, a set can be identified with a multiset taking only the values 0 and 1, and so sets are redundant. However, sets occur more commonly than multisets and are more familiar, thus we prefer to keep both.) Consider the transformation

$$domMcard : (\mu \to \Omega) \to (\mu \to Nat) \to Nat$$
$$domMcard\ b\ t = mcard\ (\lambda x.if\ (b\ x)\ then\ (t\ x)\ else\ 0),$$

where *mcard* computes the cardinality of a multiset. Thus, for each $b : \mu \to \Omega$,

$$(domMcard\ b) : (\mu \to Nat) \to Nat$$

is a function on multisets whose elements have type μ that computes the cardinality of the submultiset of elements of the argument that satisfy the predicate b.

One can then obtain a predicate by composing $(domMcard\ b)$ with, say, a predicate $(> N)$, for some $N \geq 0$. This gives the predicate $(domMcard\ b) \cdot (> N)$, which is true of a multiset iff the cardinality of the submultiset of elements satisfying the predicate b is strictly greater than N.

To obtain a second transformation on multisets, consider the following transformation.

$$codMcard : (Nat \to \Omega) \to (\mu \to Nat) \to Nat$$
$$codMcard\ c\ t = mcard\ (\lambda x.if\ (c\ (t\ x))\ then\ (t\ x)\ else\ 0).$$

Thus, for each predicate $c : Nat \to \Omega$,

$$(codMcard\ c) : (\mu \to Nat) \to Nat$$

is a function on multisets whose elements have type μ that removes elements of the multiset whose *multiplicity* does not satisfy the predicate c and then computes the cardinality of the resulting submultiset.

Note the symmetry between ($domMcard\ b$) and ($codMcard\ c$): the first filters on elements in the domain of the multiset and the second filters on elements in the codomain. (This also explains why the analogue of ($codMcard\ c$) is missing for sets. The codomain Ω is too simple to give interesting predicates.)

Multisets also support the analogue of the transformations $setExists_n$ for sets. The transformation $msetExists_n$, for $n \geq 1$, is defined as follows.

$$
\begin{aligned}
&msetExists_n : (\mu \to \Omega) \to \cdots \to (\mu \to \Omega) \to (\mu \to Nat) \to \Omega \\
&msetExists_n\ b_1 \ldots b_n\ t = \exists x_1. \cdots \exists x_n.(b_1\ x_1) \wedge \cdots \wedge (b_n\ x_n) \wedge \\
&\qquad\qquad ((t\ x_1) > 0)\ \wedge \cdots \wedge\ ((t\ x_n) > 0) \wedge (x_1 \neq x_2) \wedge \cdots \wedge (x_{n-1} \neq x_n).
\end{aligned}
$$

The analogue for multisets of $setAll$ is the transformation

$$msetAll : (\mu \to \Omega) \to (\mu \to Nat) \to \Omega$$

defined by

$$msetAll\ b\ t = \forall x.(((t\ x) > 0) \longrightarrow (b\ x)).$$

Thus ($msetAll\ b$) is true of a multiset iff every element in the multiset satisfies the predicate b.

4.4 Lists

Lists are represented using the following data constructors.

$$
\begin{aligned}
&[] : List\ \mu \\
&(:) : \mu \to List\ \mu \to List\ \mu.
\end{aligned}
$$

A list has a type $List\ \mu$, where μ is the type of the items in the list. Lists are constructed as usual from the empty list [] and the cons function (:).

The most basic transformations on lists that are useful for inductive learning are *null*, which tests whether a list is empty or not; *head* which returns the first item in a list; *tail* which returns the tail of a list; and *last* which returns the last item in a list. Given a non-negative integer N as an index and a list, the transformation ($!!N$) returns the item at that index in the list. (The index of the first item is 0.) If we are not concerned about the order of the items in a list, we can convert it to a multiset with the transformation *listToMultiset* and then apply predicates on the multiset. The multiplicity of an item in the multiset is the number of times it occurs in the list. The transformation *listToSet* returns the set of items in a list. We can also pull out sublists from the list: given an integer $N \geq 2$, ($sublists\ N$) is a transformation that returns the set of sublists of contiguous items of length N. Finally, *filterLength* takes a predicate and a list as arguments and returns the length of the list obtained from the original one by including only those items that satisfy the predicate. Here are the signatures of these transformations.

$$
\begin{aligned}
&null : List\ \mu \to \Omega \\
&head : List\ \mu \to \mu \\
&tail : List\ \mu \to List\ \mu \\
&last : List\ \mu \to \mu \\
&(!!N) : List\ \mu \to \mu \\
&listToMultiset : List\ \mu \to (\mu \to Nat) \\
&listToSet : List\ \mu \to \{\mu\} \\
&(sublists\ N) : List\ \mu \to \{List\ \mu\} \\
&filterLength : (\mu \to \Omega) \to List\ \mu \to Nat.
\end{aligned}
$$

4.5 Binary trees

Next we consider binary trees, the standard representation of which uses the following data constructors.

$$\begin{aligned}&\mathit{Null} : \mathit{BTree}\ \mu\\&\mathit{BNode} :\ \mathit{BTree}\ \mu \to \mu \to \mathit{BTree}\ \mu \to \mathit{BTree}\ \mu.\end{aligned}$$

Here $\mathit{BTree}\ \mu$ is the type of a binary tree whose nodes have type μ, *Null* represents the empty binary tree and *BNode* is a data constructor whose first argument is the left subtree, the second is the root node, and the third is the right subtree of the tree.

Some useful transformations on binary trees for making predicates are *emptyBTree*, which tests whether a tree is empty or not; *root*, which returns the root node; *leftTree*, which returns the left subtree; *rightTree*, which returns the right subtree; *nodeSet*, which returns the set of nodes in the tree; *nodeMultiset*, which returns the multiset of nodes in the tree; and *branches*, which returns the list of branches (ordered from left to right), where each branch is regarded as a list of nodes beginning with the root and ending with a leaf node. The signatures for these transformations are as follows.

$$\begin{aligned}&\mathit{emptyTree} : \mathit{BTree}\ \mu \to \Omega\\&\mathit{root} : \mathit{BTree}\ \mu \to\ \mu\\&\mathit{leftTree} : \mathit{BTree}\ \mu \to \mathit{BTree}\ \mu\\&\mathit{rightTree} : \mathit{BTree}\ \mu \to \mathit{BTree}\ \mu\\&\mathit{nodeSet} : \mathit{BTree}\ \mu \to \{\mu\}\\&\mathit{nodeMultiset} : \mathit{BTree}\ \mu \to (\mu \to \mathit{Nat})\\&\mathit{branches} : \mathit{BTree}\ \mu \to \mathit{List}\ (\mathit{List}\ \mu).\end{aligned}$$

4.6 Trees

The standard representation of a tree (where the subtrees have an order) uses the data constructor *Node*, where

$$\mathit{Node} : \mu \to \mathit{List}\ (\mathit{Tree}\ \mu) \to \mathit{Tree}\ \mu.$$

Here $\mathit{Tree}\ \mu$ is the type of a tree and *Node* is a data constructor whose first argument is the root node and second argument is the list of subtrees.

Some useful transformations on trees for making predicates are *root*, which returns the root node; *numberOfSubtrees*, which returns the number of subtrees; $(\mathit{subtree}\ N)$, which returns the Nth subtree; *nodeSet*, which returns the set of all nodes in the tree; *nodeMultiset*, which returns the multiset of all nodes in the tree; and *branches*, which returns the list of branches (in the order induced by the order of the subtrees).

$$\begin{aligned}&\mathit{root} : \mathit{Tree}\ \mu \to \mu\\&\mathit{numberOfSubtrees} : \mathit{Tree}\ \mu \to \mathit{Nat}\\&(\mathit{subtree}\ N) : \mathit{Tree}\ \mu \to \mathit{Tree}\ \mu\\&\mathit{nodeSet} : \mathit{Tree}\ \mu \to \{\mu\}\\&\mathit{nodeMultiset} : \mathit{Tree}\ \mu \to (\mu \to \mathit{Nat})\\&\mathit{branches} : \mathit{Tree}\ \mu \to \mathit{List}\ (\mathit{List}\ \mu).\end{aligned}$$

4.7 Graphs

As we have seen already, the type of an (undirected) graph is $Graph\ \nu\ \varepsilon$, where ν is the type of information in the vertices and ε is the type of information in the edges.

$$Label = Nat$$
$$Graph\ \nu\ \varepsilon = \{Label \times \nu\} \times \{(Label \to Nat) \times \varepsilon\}.$$

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 $Vertex\ \nu\ \varepsilon$ and a type constructor $Edge$ such that the type of an edge is $Edge\ \nu\ \varepsilon$. The following transformations on graphs are provided.

$$vertices : Graph\ \nu\ \varepsilon \to \{Vertex\ \nu\ \varepsilon\}$$
$$edges : Graph\ \nu\ \varepsilon \to \{Edge\ \nu\ \varepsilon\}$$
$$vertex : Vertex\ \nu\ \varepsilon \to \nu$$
$$edge : Edge\ \nu\ \varepsilon \to \varepsilon$$
$$connects : Edge\ \nu\ \varepsilon \to (Vertex\ \nu\ \varepsilon \to Nat)$$
$$edgesAtVertex : Vertex\ \nu\ \varepsilon \to \{Edge\ \nu\ \varepsilon\}$$
$$(subgraphs\ N) : Graph\ \nu\ \varepsilon \to \{Graph\ \nu\ \varepsilon\}.$$

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, *edgesAtVertex* returns the set of edges all having some vertex at one end, and ($subgraphs\ N$) returns the set of (connected) subgraphs of the graph that contain N vertices.

4.8 Digraphs

The type of a directed graph is $Digraph\ \nu\ \varepsilon$, where ν is the type of information in the vertices and ε is the type of information in the edges, and

$$Digraph\ \nu\ \varepsilon = \{Label \times \nu\} \times \{(Label \times Label) \times E\}.$$

There is a type constructor $DiVertex$ such that the type of a vertex is $DiVertex\ \nu\ \varepsilon$ and a type constructor $DiEdge$ such that the type of an edge is $DiEdge\ \nu\ \varepsilon$. There are also the following transformations.

$$vertices : Digraph\ \nu\ \varepsilon \to \{DiVertex\ \nu\ \varepsilon\}$$
$$edges : Digraph\ \nu\ \varepsilon \to \{DiEdge\ \nu\ \varepsilon\}$$
$$vertex : DiVertex\ \nu\ \varepsilon \to \nu$$
$$edge : DiEdge\ \nu\ \varepsilon \to \varepsilon$$
$$connects : DiEdge\ \nu\ \varepsilon \to DiVertex\ \nu\ \varepsilon \times DiVertex\ \nu\ \varepsilon$$
$$edgesToVertex : DiVertex\ \nu\ \varepsilon \to \{DiEdge\ \nu\ \varepsilon\}$$
$$edgesFromVertex : DiVertex\ \nu\ \varepsilon \to \{DiEdge\ \nu\ \varepsilon\}$$
$$(subgraphs\ N) : Digraph\ \nu\ \varepsilon \to \{Digraph\ \nu\ \varepsilon\}.$$

The transformation *vertices* returns the set of vertices of a digraph, *edges* returns the set of edges of a digraph, *vertex* returns the information at a vertex, *edge* returns the information on an edge, *connects* returns the (ordered) pair of vertices joined by an edge, *edgesToVertex* returns the set of edges that are directed into some vertex, *edgesFromVertex* returns the set of edges that are directed away from some vertex, and ($subgraphs\ N$) returns the set of (connected) subdigraphs of the digraph that contain N vertices.

5 Ten Easy Pieces

We now present a series of applications which illustrate the ideas that have been introduced. The applications are deliberately simple ones and are varied, so that the approach taken here to knowledge representation can be more easily comprehended. We note here that the accuracy (on the training examples) of the induced definitions in this section are all 100%, although this is of no importance since these applications are small and artificial. Also it is not explained in this section how the search for suitable predicates and induced definitions is conducted – this is postponed to the next section. Primarily, we are interested in this section in knowledge representation issues and the form of the induced definitions for the various types.

5.1 Tuples

In attribute-value learning, individuals are, in effect, tuples of constants. In our context, this implies the definition of a tuple type, which is the domain of the function to be learned, and a base type for each attribute.

Our example is a typical attribute-value problem which involves learning the concept of playing, or not playing, tennis, according to the weather [Qui86], [Mit97, p.59]. First we introduce the types *Outlook*, *Temperature*, *Humidity*, and *Wind* as follows.

$$\begin{aligned}
&\mathit{Sunny}, \mathit{Overcast}, \mathit{Rain} : \mathit{Outlook}\\
&\mathit{Hot}, \mathit{Mild}, \mathit{Cool} : \mathit{Temperature}\\
&\mathit{High}, \mathit{Normal}, \mathit{Low} : \mathit{Humidity}\\
&\mathit{Strong}, \mathit{Medium}, \mathit{Weak} : \mathit{Wind}.
\end{aligned}$$

It will be convenient to introduce the following type synonym.

$$\mathit{Weather} = \mathit{Outlook} \times \mathit{Temperature} \times \mathit{Humidity} \times \mathit{Wind}.$$

We want to learn the definition of the function *playTennis* having signature as follows.

$$\mathit{playTennis} : \mathit{Weather} \to \Omega.$$

Here are the examples.

$$\begin{aligned}
&\mathit{playTennis}\ (\mathit{Overcast}, \mathit{Hot}, \mathit{High}, \mathit{Weak}) = \top\\
&\mathit{playTennis}\ (\mathit{Rain}, \mathit{Mild}, \mathit{High}, \mathit{Weak}) = \top\\
&\mathit{playTennis}\ (\mathit{Rain}, \mathit{Cool}, \mathit{Normal}, \mathit{Weak}) = \top\\
&\mathit{playTennis}\ (\mathit{Overcast}, \mathit{Cool}, \mathit{Normal}, \mathit{Strong}) = \top\\
&\mathit{playTennis}\ (\mathit{Sunny}, \mathit{Cool}, \mathit{Normal}, \mathit{Weak}) = \top\\
&\mathit{playTennis}\ (\mathit{Rain}, \mathit{Mild}, \mathit{Normal}, \mathit{Weak}) = \top\\
&\mathit{playTennis}\ (\mathit{Sunny}, \mathit{Mild}, \mathit{Normal}, \mathit{Strong}) = \top\\
&\mathit{playTennis}\ (\mathit{Overcast}, \mathit{Mild}, \mathit{High}, \mathit{Strong}) = \top\\
&\mathit{playTennis}\ (\mathit{Overcast}, \mathit{Hot}, \mathit{Normal}, \mathit{Weak}) = \top\\
&\mathit{playTennis}\ (\mathit{Sunny}, \mathit{Hot}, \mathit{High}, \mathit{Weak}) = \bot\\
&\mathit{playTennis}\ (\mathit{Sunny}, \mathit{Hot}, \mathit{High}, \mathit{Strong}) = \bot\\
&\mathit{playTennis}\ (\mathit{Rain}, \mathit{Cool}, \mathit{Normal}, \mathit{Strong}) = \bot\\
&\mathit{playTennis}\ (\mathit{Sunny}, \mathit{Mild}, \mathit{High}, \mathit{Weak}) = \bot\\
&\mathit{playTennis}\ (\mathit{Rain}, \mathit{Mild}, \mathit{High}, \mathit{Strong}) = \bot.
\end{aligned}$$

The hypothesis language contains the following transformations.

$$
\begin{aligned}
&(= \mathit{Sunny}) : \mathit{Outlook} \to \Omega \\
&(= \mathit{Overcast}) : \mathit{Outlook} \to \Omega \\
&(= \mathit{Rain}) : \mathit{Outlook} \to \Omega \\
&(= \mathit{Hot}) : \mathit{Temperature} \to \Omega \\
&(= \mathit{Mild}) : \mathit{Temperature} \to \Omega \\
&(= \mathit{Cool}) : \mathit{Temperature} \to \Omega \\
&(= \mathit{High}) : \mathit{Humidity} \to \Omega \\
&(= \mathit{Normal}) : \mathit{Humidity} \to \Omega \\
&(= \mathit{Low}) : \mathit{Humidity} \to \Omega \\
&(= \mathit{Strong}) : \mathit{Wind} \to \Omega \\
&(= \mathit{Medium}) : \mathit{Wind} \to \Omega \\
&(= \mathit{Weak}) : \mathit{Wind} \to \Omega \\
&\mathit{projOutlook} : \mathit{Weather} \to \mathit{Outlook} \\
&\mathit{projTemperature} : \mathit{Weather} \to \mathit{Temperature} \\
&\mathit{projHumidity} : \mathit{Weather} \to \mathit{Humidity} \\
&\mathit{projWind} : \mathit{Weather} \to \mathit{Wind} \\
&\wedge_2 : (\mathit{Weather} \to \Omega) \to (\mathit{Weather} \to \Omega) \to \mathit{Weather} \to \Omega \\
&\wedge_3 : (\mathit{Weather} \to \Omega) \to (\mathit{Weather} \to \Omega) \to (\mathit{Weather} \to \Omega) \to \mathit{Weather} \to \Omega \\
&\wedge_4 : (\mathit{Weather} \to \Omega) \to (\mathit{Weather} \to \Omega) \to (\mathit{Weather} \to \Omega) \to (\mathit{Weather} \to \Omega) \to \\
&\hspace{20em} \mathit{Weather} \to \Omega.
\end{aligned}
$$

Here *projOutlook* is the projection from *Weather* onto the component *Outlook*. Similarly, for *projTemperature*, *projHumidity*, and *projWind*.

In this setting, our learning system found the following definition for the function *playTennis*.

$$
\begin{aligned}
&\mathit{playTennis}\ w = \\
&\quad \mathit{if}\ \ \wedge_2\ (\mathit{projOutlook} \cdot (= \mathit{Sunny}))\ (\mathit{projHumidity} \cdot (= \mathit{High}))\ w \\
&\quad \mathit{then}\ \bot \\
&\quad \mathit{else\ if}\ \ \wedge_2\ (\mathit{projOutlook} \cdot (= \mathit{Rain}))\ (\mathit{projWind} \cdot (= \mathit{Strong}))\ w \\
&\qquad \mathit{then}\ \bot \\
&\qquad \mathit{else}\ \top.
\end{aligned}
$$

"The weather is suitable for playing tennis unless the outlook is sunny and the humidity is high or the outlook is rain and the wind is strong".

5.2 Sets

We now turn to a problem which is interesting because it illustrates the multiple-instance problem that is discussed in [DLLP97]. From a knowledge representation point of view, a multiple-instance problem is indicated by having a term which is a set at the top level representing an individual.

Consider the problem of determining whether a key in a bunch of keys can open a door. (This problem was suggested in [DLLP97].) More precisely, suppose there are a number of 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 is prototypical of a number of important practical problems such as that of pharmacophore discovery [DLLP97, FMPS98].

A characteristic property of the multiple-instance problem is that the "individuals" are actually a collection of similar entities and the problem is to try to find a suitable entity in each collection satisfying some predicate. Thus the kind of predicate we want to find has the form

$$(setExists_1\ b),$$

for a suitable predicate b on the entities in the collection. Expressed this way, it is clear that the multiple-instance problem is just a special case of the general framework that is developed in this paper in which the type of the individuals is a set. Since the individuals to be classified have a set type, following the principles proposed here, we should go inside the set to apply predicates to its elements.

To illustrate the multiple-instance problem, consider the following application concerning bunches of keys.

$$\begin{aligned}
&Abloy, Chubb, Rubo, Yale : Make\\
&Short, Medium, Long : Length\\
&Narrow, Normal, Broad : Width.
\end{aligned}$$

We introduce the following type synonyms.

$$\begin{aligned}
&NumProngs = Nat\\
&Key = Make \times NumProngs \times Length \times Width\\
&Bunch = \{Key\}.
\end{aligned}$$

We want to learn the following function.

$$opens : Bunch \to \Omega.$$

The examples are as follows.

$$\begin{aligned}
&opens\ \{(Abloy, 3, Short, Normal), (Abloy, 4, Medium, Broad),\\
&\qquad\qquad (Chubb, 3, Long, Narrow)\} = \top\\
&opens\ \{(Abloy, 3, Medium, Broad), (Chubb, 2, Long, Normal),\\
&\qquad\qquad (Chubb, 4, Medium, Broad)\} = \top\\
&opens\ \{(Abloy, 3, Short, Broad), (Abloy, 4, Medium, Broad),\\
&\qquad\qquad (Chubb, 3, Long, Narrow)\} = \top\\
&opens\ \{(Abloy, 3, Medium, Broad), (Abloy, 4, Medium, Narrow),\\
&\qquad\qquad (Chubb, 3, Long, Broad), (Yale, 4, Medium, Broad)\} = \top\\
&opens\ \{(Abloy, 3, Medium, Narrow), (Chubb, 6, Medium, Normal),\\
&\qquad\qquad (Rubo, 5, Short, Narrow), (Yale, 4, Long, Broad)\} = \top\\
&opens\ \{(Chubb, 3, Short, Broad), (Chubb, 4, Medium, Broad), (Yale, 3, Short, Narrow),\\
&\qquad\qquad (Yale, 4, Long, Normal)\} = \bot\\
&opens\ \{(Yale, 3, Long, Narrow), (Yale, 4, Long, Broad)\} = \bot\\
&opens\ \{(Abloy, 3, Short, Broad), (Chubb, 3, Short, Broad), (Rubo, 4, Long, Broad),\\
&\qquad\qquad (Yale, 4, Long, Broad)\} = \bot\\
&opens\ \{(Abloy, 4, Short, Broad), (Chubb, 3, Medium, Broad),\\
&\qquad\qquad (Rubo, 5, Long, Narrow)\} = \bot.
\end{aligned}$$

The hypothesis language contains the following transformations.

$$
\begin{aligned}
&(= \mathit{Abloy}) : \mathit{Make} \to \Omega \\
&(= \mathit{Chubb}) : \mathit{Make} \to \Omega \\
&(= \mathit{Rubo}) : \mathit{Make} \to \Omega \\
&(= \mathit{Yale}) : \mathit{Make} \to \Omega \\
&(= 2) : \mathit{NumProngs} \to \Omega \\
&(= 3) : \mathit{NumProngs} \to \Omega \\
&(= 4) : \mathit{NumProngs} \to \Omega \\
&(= 5) : \mathit{NumProngs} \to \Omega \\
&(= 6) : \mathit{NumProngs} \to \Omega \\
&(= \mathit{Short}) : \mathit{Length} \to \Omega \\
&(= \mathit{Medium}) : \mathit{Length} \to \Omega \\
&(= \mathit{Long}) : \mathit{Length} \to \Omega \\
&(= \mathit{Narrow}) : \mathit{Width} \to \Omega \\
&(= \mathit{Normal}) : \mathit{Width} \to \Omega \\
&(= \mathit{Broad}) : \mathit{Width} \to \Omega \\
&\mathit{projMake} : \mathit{Key} \to \mathit{Make} \\
&\mathit{projNumProngs} : \mathit{Key} \to \mathit{NumProngs} \\
&\mathit{projLength} : \mathit{Key} \to \mathit{Length} \\
&\mathit{projWidth} : \mathit{Key} \to \mathit{Width} \\
&\mathit{setExists}_1 : (\mathit{Key} \to \Omega) \to \mathit{Bunch} \to \Omega \\
&\wedge_2 : (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to \mathit{Key} \to \Omega \\
&\wedge_3 : (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to \mathit{Key} \to \Omega \\
&\wedge_4 : (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to \mathit{Key} \to \Omega.
\end{aligned}
$$

Our learning system found the following definition for the function *opens*.

$$
\begin{aligned}
&\mathit{opens}\ b = \\
&\qquad \mathit{if}\ \mathit{setExists}_1\ (\wedge_2\ (\mathit{projMake} \cdot (= \mathit{Abloy}))\ (\mathit{projLength} \cdot (= \mathit{Medium})))\ b \\
&\qquad \mathit{then}\ \top \\
&\qquad \mathit{else}\ \bot.
\end{aligned}
$$

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

Needless to say, the definition for *opens* can be simplified to

$$\mathit{opens}\ b = \mathit{setExists}_1\ (\wedge_2\ (\mathit{projMake} \cdot (= \mathit{Abloy}))\ (\mathit{projLength} \cdot (= \mathit{Medium})))\ b.$$

5.3 Multisets

We now consider an application which illustrates the use of multisets for knowledge representation.

Consider the problem of trying to decide whether a climate in some country is pleasant or not. The climate is modelled by a multiset. Each item in a multiset is a 4-tuple characterising the main features of the weather during a day and the multiplicity of the item is the number of times during a year a day with those particular weather features occurs. Thus, using the type *Weather* from Subsection 5.1, we can model a climate by the following multiset type.

$$\mathit{Climate} = \mathit{Weather} \to \mathit{Nat}.$$

The function we want to learn is *pleasant* whose signature is as follows.

$$pleasant : Climate \to \Omega.$$

Suppose the examples are as follows.

$$\begin{aligned}
&pleasant\ (\lambda w.if\ w = (Sunny, Mild, Normal, Strong)\ then\ 220\ else\\
&\qquad if\ w = (Overcast, Cool, Normal, Strong)\ then\ 15\ else\\
&\qquad if\ w = (Sunny, Hot, High, Strong)\ then\ 100\ else\\
&\qquad if\ w = (Rain, Cool, Normal, Strong)\ then\ 30\ else\ 0) = \top\\
&pleasant\ (\lambda w.if\ w = (Sunny, Mild, High, Strong)\ then\ 180\ else\\
&\qquad if\ w = (Sunny, Mild, High, Weak)\ then\ 25\ else\\
&\qquad if\ w = (Rain, Cool, Normal, Weak)\ then\ \ 160\ else\ 0) = \top\\
&pleasant\ (\lambda w.if\ w = (Sunny, Mild, High, Strong)\ then\ 135\ else\\
&\qquad if\ w = (Rain, Cool, Normal, Strong)\ then\ 130\ else\\
&\qquad if\ w = (Sunny, Mild, Normal, Weak)\ then\ 100\ else\ 0) = \top\\
&pleasant\ (\lambda w.if\ w = (Rain, Mild, Normal, Weak)\ then\ 150\ else\\
&\qquad if\ w = (Sunny, Mild, High, Weak)\ then\ 215\ else\ 0) = \top\\
&pleasant\ (\lambda w.if\ w = (Overcast, Hot, High, Weak)\ then\ 99\ else\\
&\qquad if\ w = (Overcast, Cool, Normal, Strong)\ then\ 266\ else\ 0) = \bot\\
&pleasant\ (\lambda w.if\ w = (Overcast, Hot, High, Weak)\ then\ 35\ else\\
&\qquad if\ w = (Overcast, Cool, Normal, Strong)\ then\ 124\ else\\
&\qquad if\ w = (Rain, Cool, Normal, Strong)\ then\ 206\ else\ 0) = \bot\\
&pleasant\ (\lambda w.if\ w = (Rain, Mild, High, Weak)\ then\ 160\ else\\
&\qquad if\ w = (Overcast, Cool, Normal, Strong)\ then\ 50\ else\\
&\qquad if\ w = (Rain, Cool, Normal, Weak)\ then\ 155\ else\ 0) = \bot\\
&pleasant\ (\lambda w.if\ w = (Sunny, Mild, High, Weak)\ then\ 135\ else\\
&\qquad if\ w = (Rain, Cool, Normal, Strong)\ then\ 34\ else\\
&\qquad if\ w = (Sunny, Hot, High, Strong)\ then\ 196\ else\ 0) = \bot\\
&pleasant\ (\lambda w.if\ w = (Rain, Mild, High, Strong)\ then\ 100\ else\\
&\qquad if\ w = (Overcast, Hot, Normal, Weak)\ then\ 80\ else\\
&\qquad if\ w = (Rain, Cool, Normal, Weak)\ then\ 185\ else\ 0) = \bot.
\end{aligned}$$

The hypothesis language contains the following transformations.

$$\begin{aligned}
&(= Sunny) : Outlook \to \Omega\\
&\qquad\vdots\\
&(= Weak) : Wind \to \Omega\\
&projOutlook : Weather \to Outlook\\
&projTemperature : Weather \to Temperature\\
&projHumidity : Weather \to Humidity\\
&projWind : Weather \to Wind\\
&(> 0) : Nat \to \Omega\\
&(> 50) : Nat \to \Omega\\
&(> 100) : Nat \to \Omega
\end{aligned}$$

$$\begin{aligned}
&\vdots\\
&(>350) : Nat \to \Omega\\
&domMcard : (Weather \to \Omega) \to Climate \to Nat\\
&\wedge_2 : (Weather \to \Omega) \to (Weather \to \Omega) \to Weather \to \Omega\\
&\wedge_3 : (Weather \to \Omega) \to (Weather \to \Omega) \to (Weather \to \Omega) \to Weather \to \Omega\\
&\wedge_4 : (Weather \to \Omega) \to (Weather \to \Omega) \to (Weather \to \Omega) \to (Weather \to \Omega) \to\\
&\qquad\qquad Weather \to \Omega.
\end{aligned}$$

Our learning system found the following definition for the function *pleasant*.

$$\begin{aligned}
&pleasant\ c =\\
&\quad if\ (domMcard\ (\wedge_2\ (projOutlook \cdot (= Sunny))\ (projTemperature \cdot (= Mild)))) \cdot (>150)\ c\\
&\quad then\ \top\\
&\quad else\ \bot.
\end{aligned}$$

"A country has a pleasant climate if and only if it has at least 150 days in a year that have a sunny outlook and a mild temperature".

5.4 Multisets again

As another illustration of the use of multisets, consider the problem of text categorisation. For many purposes in information retrieval it is appropriate to regard a document as a multiset of words, where the multiplicity of a word is the number of times it occurs in the document.

Consider now the problem of inducing a classification function for newspaper headlines, where headlines can be either financial, political, or miscellaneous. For this purpose, we introduce the type *Class* and associated data constructors.

$$Finance, Politics, Miscellaneous : Class.$$

A headline is a multiset of words and a word is a string.

$$\begin{aligned}
&Word = String\\
&Headline = Word \to Nat.
\end{aligned}$$

We want to learn the definition of the function *classify* having signature as follows.

$$classify : Headline \to Class.$$

Here are the examples. To reduce the notational overhead, we let $\langle word_1, word_2, \ldots\rangle$ denote the multiset containing the words $word_1$, and so on, where a word appears in the expression as many times as its multiplicity.

$$\begin{aligned}
&classify\ \langle\text{“}hospital\text{”}, \text{“}enquiry\text{”}, \text{“}defies\text{”}, \text{“}prime\text{”}, \text{“}minister\text{”}\rangle = Politics\\
&classify\ \langle\text{“}prime\text{”}, \text{“}minister\text{”}, \text{“}gambles\text{”}, \text{“}on\text{”}, \text{“}coming\text{”}, \text{“}clean\text{”}\rangle = Politics\\
&classify\ \langle\text{“}prime\text{”}, \text{“}minister\text{”}, \text{“}shifts\text{”}, \text{“}to\text{”}, \text{“}overdrive\text{”}\rangle = Politics\\
&classify\ \langle\text{“}parliament\text{”}, \text{“}expects\text{”}, \text{“}prime\text{”}, \text{“}minister\text{”}, \text{“}to\text{”}, \text{“}depart\text{”}\rangle = Politics\\
&classify\ \langle\text{“}washington\text{”}, \text{“}launches\text{”}, \text{“}currency\text{”}, \text{“}war\text{”}, \text{“}by\text{”}, \text{“}stealth\text{”}\rangle = Finance\\
&classify\ \langle\text{“}signs\text{”}, \text{“}of\text{”}, \text{“}life\text{”}, \text{“}beyond\text{”}, \text{“}currency\text{”}, \text{“}crisis\text{”}\rangle = Finance\\
&classify\ \langle\text{“}tax\text{”}, \text{“}to\text{”}, \text{“}drop\text{”}, \text{“}next\text{”}, \text{“}tax\text{”}, \text{“}year\text{”}\rangle = Finance\\
&classify\ \langle\text{“}parliament\text{”}, \text{“}debates\text{”}, \text{“}tax\text{”}, \text{“}law\text{”}\rangle = Finance\\
&classify\ \langle\text{“}time\text{”}, \text{“}for\text{”}, \text{“}a\text{”}, \text{“}deal\text{”}, \text{“}on\text{”}, \text{“}tax\text{”}, \text{“}reform\text{”}\rangle = Finance
\end{aligned}$$

$$
\begin{aligned}
&classify\ \langle \text{“}tax\text{”}, \text{“}due\text{”}, \text{“}to\text{”}, \text{“}go\text{”}, \text{“}down\text{”} \rangle = Finance\\
&classify\ \langle \text{“}currency\text{”}, \text{“}crisis\text{”}, \text{“}hits\text{”}, \text{“}asia\text{”} \rangle = Finance\\
&classify\ \langle \text{“}australian\text{”}, \text{“}currency\text{”}, \text{“}problems\text{”}, \text{“}fade\text{”} \rangle = Finance\\
&classify\ \langle \text{“}europe\text{”}, \text{“}to\text{”}, \text{“}address\text{”}, \text{“}currency\text{”}, \text{“}crisis\text{”} \rangle = Finance\\
&classify\ \langle \text{“}prime\text{”}, \text{“}time\text{”}, \text{“}to\text{”}, \text{“}invest\text{”}, \text{“}in\text{”}, \text{“}currency\text{”}, \text{“}exchange\text{”} \rangle = Finance\\
&classify\ \langle \text{“}minister\text{”}, \text{“}announces\text{”}, \text{“}massive\text{”}, \text{“}tax\text{”}, \text{“}cut\text{”} \rangle = Finance\\
&classify\ \langle \text{“}flooded\text{”}, \text{“}asia\text{”}, \text{“}counts\text{”}, \text{“}the\text{”}, \text{“}cost\text{”} \rangle = Miscellaneous\\
&classify\ \langle \text{“}kasparov\text{”}, \text{“}moves\text{”}, \text{“}in\text{”}, \text{“}on\text{”}, \text{“}world\text{”} \rangle = Miscellaneous\\
&classify\ \langle \text{“}Y2K\text{”}, \text{“}avoids\text{”}, \text{“}Y2K\text{”}, \text{“}bug\text{”} \rangle = Miscellaneous\\
&classify\ \langle \text{“}minister\text{”}, \text{“}bites\text{”}, \text{“}dog\text{”} \rangle = Miscellaneous\\
&classify\ \langle \text{“}enqvist\text{”}, \text{“}elbowed\text{”}, \text{“}out\text{”} \rangle = Miscellaneous\\
&classify\ \langle \text{“}dokic\text{”}, \text{“}saves\text{”}, \text{“}biggest\text{”}, \text{“}serve\text{”}, \text{“}for\text{”}, \text{“}last\text{”} \rangle = Miscellaneous.
\end{aligned}
$$

The hypothesis language contains the following transformations.

$$
\begin{aligned}
&(= \text{“}hospital\text{”}) : Word \to \Omega\\
&\qquad\vdots\\
&(= \text{“}last\text{”}) : Word \to \Omega\\
&msetExists_1 : (Word \to \Omega) \to Headline \to \Omega\\
&msetExists_2 : (Word \to \Omega) \to (Word \to \Omega) \to Headline \to \Omega.
\end{aligned}
$$

Our learning system found the following definition for the function *classify*.

$$
\begin{aligned}
&classify\ h =\\
&\quad \mathit{if}\ msetExists_2\ (= \text{“}minister\text{”})\ (= \text{“}prime\text{”})\ h\\
&\quad \mathit{then}\ Politics\\
&\quad \mathit{else\ if}\ msetExists_1\ (= \text{“}currency\text{”})\ h\\
&\qquad \mathit{then}\ Finance\\
&\qquad \mathit{else\ if}\ msetExists_1\ (= \text{“}tax\text{”})\ h\\
&\qquad\quad \mathit{then}\ Finance\\
&\qquad\quad \mathit{else}\ Miscellaneous.
\end{aligned}
$$

“If a headline contains the words ‘minister’ and ‘prime’, then it is political; else if it contains the word ‘currency’, then it is financial; else if it contains the word ‘tax’, then it is financial; else it is miscellaneous”.

5.5 Lists

The next application is the well-known East-West Challenge of Michalski [ML77, MP94, Eas], which consists of determining whether a train is headed west or east based on a number of its characteristics. The trains are illustrated in Figure 2.

The most natural type to model a train is a list. We first introduce the types *Direction*, *Shape*,

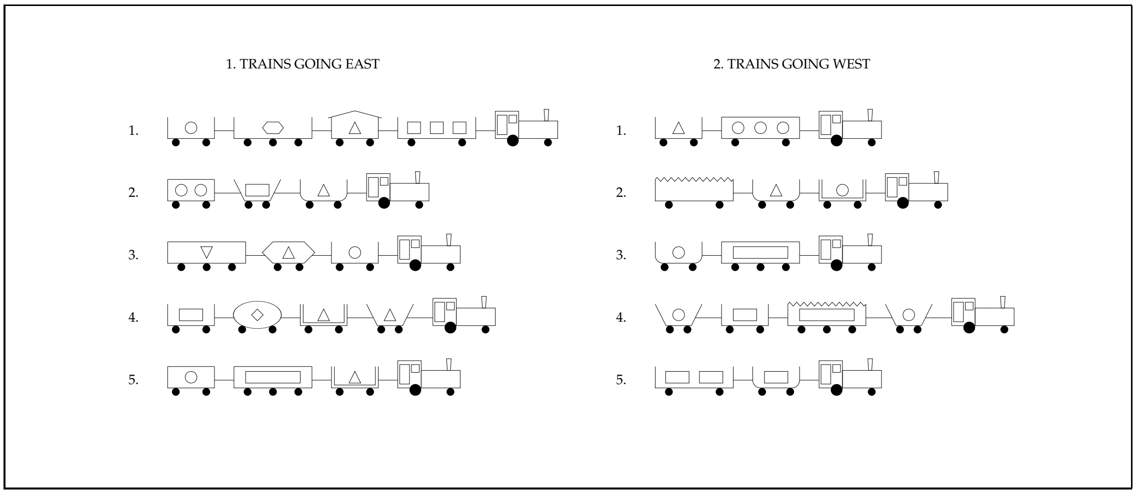

Figure 2: Trains going east and west

Length, *Kind*, *Roof*, and *Object*.

$$\begin{array}{l}
\mathit{East}, \mathit{West} : \mathit{Direction}\\
\mathit{Rectangular}, \mathit{DoubleRectangular}, \mathit{UShaped}, \mathit{BucketShaped}, \mathit{Hexagonal}, \mathit{Ellipsoidal} : \mathit{Shape}\\
\mathit{Long}, \mathit{Short} : \mathit{Length}\\
\mathit{Closed}, \mathit{Open} : \mathit{Kind}\\
\mathit{Flat}, \mathit{Jagged}, \mathit{Peaked}, \mathit{Curved}, \mathit{None} : \mathit{Roof}\\
\mathit{Circle}, \mathit{Hexagon}, \mathit{Square}, \mathit{Rectangle}, \mathit{LongRectangle}, \mathit{Triangle}, \mathit{InvertedTriangle},\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Diamond}, \mathit{Null} : \mathit{Object}.
\end{array}$$

It will be convenient to introduce the following type synonyms.

$$\begin{array}{l}
\mathit{NumWheels} = \mathit{Nat}\\
\mathit{NumObjects} = \mathit{Nat}\\
\mathit{Load} = \mathit{Object} \times \mathit{NumObjects}\\
\mathit{Car} = \mathit{Shape} \times \mathit{Length} \times \mathit{NumWheels} \times \mathit{Kind} \times \mathit{Roof} \times \mathit{Load}\\
\mathit{Train} = \mathit{List}\ \mathit{Car}.
\end{array}$$

We want to learn the definition of the function *direction* having signature as follows.

$$\mathit{direction} : \mathit{Train} \rightarrow \mathit{Direction}.$$

Here are the examples. (See Figure 2.)

$$\begin{array}{l}
\mathit{direction}\ [(\mathit{Rectangular}, \mathit{Long}, 2, \mathit{Open}, \mathit{None}, (\mathit{Square}, 3)),\\
\qquad\qquad (\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Closed}, \mathit{Peaked}, (\mathit{Triangle}, 1)),\\
\qquad\qquad (\mathit{Rectangular}, \mathit{Long}, 3, \mathit{Open}, \mathit{None}, (\mathit{Hexagon}, 1)),\\
\qquad\qquad (\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Circle}, 1))] = \mathit{East}\\
\mathit{direction}\ [(\mathit{UShaped}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Triangle}, 1)),\\
\qquad\qquad (\mathit{BucketShaped}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Rectangle}, 1)),\\
\qquad\qquad (\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Closed}, \mathit{Flat}, (\mathit{Circle}, 2))] = \mathit{East}\\
\mathit{direction}\ [(\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Circle}, 1)),
\end{array}$$

$$
\begin{aligned}
&(Hexagonal, Short, 2, Closed, Flat, (Triangle, 1)),\\
&(Rectangular, Long, 3, Closed, Flat, (InvertedTriangle, 1))] = East\\
&direction\ [(BucketShaped, Short, 2, Open, None, (Triangle, 1)),\\
&(DoubleRectangular, Short, 2, Open, None, (Triangle, 1)),\\
&(Ellipsoidal, Short, 2, Closed, Curved, (Diamond, 1)),\\
&(Rectangular, Short, 2, Open, None, (Rectangle, 1))] = East\\
&direction\ [(DoubleRectangular, Short, 2, Open, None, (Triangle, 1)),\\
&(Rectangular, Long, 3, Closed, Flat, (LongRectangle, 1)),\\
&(Rectangular, Short, 2, Closed, Flat, (Circle, 1))] = East\\
&direction\ [(Rectangular, Long, 2, Closed, Flat, (Circle, 3)),\\
&(Rectangular, Short, 2, Open, None, (Triangle, 1))] = West\\
&direction\ [(DoubleRectangular, Short, 2, Open, None, (Circle, 1)),\\
&(UShaped, Short, 2, Open, None, (Triangle, 1)),\\
&(Rectangular, Long, 2, Closed, Jagged, (Null, 0))] = West\\
&direction\ [(Rectangular, Long, 3, Closed, Flat, (LongRectangle, 1)),\\
&(UShaped, Short, 2, Open, None, (Circle, 1))] = West\\
&direction\ [(BucketShaped, Short, 2, Open, None, (Circle, 1)),\\
&(Rectangular, Long, 3, Closed, Jagged, (LongRectangle, 1)),\\
&(Rectangular, Short, 2, Open, None, (Rectangle, 1)),\\
&(BucketShaped, Short, 2, Open, None, (Circle, 1))] = West\\
&direction\ [(UShaped, Short, 2, Open, None, (Rectangle, 1)),\\
&(Rectangular, Long, 2, Open, None, (Rectangle, 2))] = West.
\end{aligned}
$$

The hypothesis language contains the following transformations.

$$
\begin{aligned}
&(= Rectangular) : Shape \to \Omega\\
&\qquad\qquad \vdots\\
&(= Null) : Object \to \Omega\\
&projShape : Car \to Shape\\
&projLength : Car \to Length\\
&projNumWheels : Car \to NumWheels\\
&projKind : Car \to Kind\\
&projRoof : Car \to Roof\\
&projLoad : Car \to Load\\
&projObject : Load \to Object\\
&projNumObjects : Load \to NumObjects\\
&null : Train \to \Omega\\
&head : Train \to Car\\
&last : Train \to Car\\
&listToSet : Train \to \{Car\}\\
&setExists_1 : (Car \to \Omega) \to \{Car\} \to \Omega\\
&\wedge_2 : (Car \to \Omega) \to (Car \to \Omega) \to Car \to \Omega\\
&\wedge_3 : (Car \to \Omega) \to (Car \to \Omega) \to (Car \to \Omega) \to Car \to \Omega.
\end{aligned}
$$

Our learning system found the following definition for the function *direction*.

$$
\begin{aligned}
&\mathit{direction}\ t = \\
&\qquad \mathit{if}\ \mathit{listToSet} \cdot (\mathit{setExists}_1\ (\wedge_2\ (\mathit{projLength} \cdot (= \mathit{Short}))\ (\mathit{projKind} \cdot (= \mathit{Closed}))))\ t \\
&\qquad \mathit{then}\ \mathit{East} \\
&\qquad \mathit{else}\ \mathit{West}.
\end{aligned}
$$

"A train is eastbound if and only if it has a short closed car".

5.6 Lists again

As a further example of the use of lists, consider the problem of classifying proteins from their primary structure (that is, sequence of amino acids).

First, we need a type for the 20 amino acids.

$$A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y : \mathit{AminoAcid}.$$

We consider a protein to be a list of amino acids.

$$\mathit{Protein} = \mathit{List}\ \mathit{AminoAcid}.$$

We want to learn the definition of the function *irregular* having signature as follows.

$$\mathit{irregular} : \mathit{Protein} \to \Omega.$$

Here are the examples.

$$
\begin{aligned}
&\mathit{irregular}\ [N, S, T, C, K, L, S, A, A, D, N, L, K, V, N, P, Q, Q] = \top \\
&\mathit{irregular}\ [V, C, V, L, S, A, A, D, N, L, M, Q, T, L, N, D, N] = \top \\
&\mathit{irregular}\ [N, D, N, L, K, V, V, C, V, L, D, D, M, Q, T, L, N, S, A, A, D, N, L, N, D, N] = \top \\
&\mathit{irregular}\ [K, V, I, E, K, A, N, N, L, D, D, N, S, A, A, D, N, L, L, N, L, N, D] = \top \\
&\mathit{irregular}\ [L, D, S, A, A, M, I, L, L, N, L, N, D] = \top \\
&\mathit{irregular}\ [A, A, D, L, E, D, D, M, Q, T, L, N, D, N, L, K, V, I, E, K, A, N, N, S, K, A] = \bot \\
&\mathit{irregular}\ [S, A, L, W, Q, F, N, G, M, I, K, C, K, I, P] = \bot \\
&\mathit{irregular}\ [N, D, N, L, K, V, I, Q, F, N, G, M, I, K, C, E, K, A, N, N, E, K, A, A] = \bot \\
&\mathit{irregular}\ [N, A, D, D, M, Q, T, L, N, L, N, D, N, L, K, V, I, E, K, S, A, N, N] = \bot \\
&\mathit{irregular}\ [M, Q, T, L, N, E, K, S, A, D, M] = \bot.
\end{aligned}
$$

The hypothesis language contains the following transformations.

$$
\begin{aligned}
&(= A) : \mathit{AminoAcid} \to \Omega \\
&\qquad\quad \vdots \\
&(= Y) : \mathit{AminoAcid} \to \Omega \\
&\mathit{null} : \mathit{Protein} \to \Omega \\
&\mathit{head} : \mathit{Protein} \to \mathit{AminoAcid} \\
&\mathit{tail} : \mathit{Protein} \to \mathit{Protein} \\
&\mathit{last} : \mathit{Protein} \to \mathit{AminoAcid} \\
&(!!0) : \mathit{Protein} \to \mathit{AminoAcid} \\
&(!!1) : \mathit{Protein} \to \mathit{AminoAcid} \\
&(!!2) : \mathit{Protein} \to \mathit{AminoAcid}
\end{aligned}
$$

$$
\begin{aligned}
&listToSet : Protein \rightarrow \{AminoAcid\}\\
&(sublists\ 3) : Protein \rightarrow \{Protein\}\\
&setExists_1 : (AminoAcid \rightarrow \Omega) \rightarrow \{AminoAcid\} \rightarrow \Omega\\
&setExists_1 : (Protein \rightarrow \Omega) \rightarrow \{Protein\} \rightarrow \Omega\\
&\wedge_2 : (Protein \rightarrow \Omega) \rightarrow (Protein \rightarrow \Omega) \rightarrow Protein \rightarrow \Omega\\
&\wedge_3 : (Protein \rightarrow \Omega) \rightarrow (Protein \rightarrow \Omega) \rightarrow (Protein \rightarrow \Omega) \rightarrow Protein \rightarrow \Omega.
\end{aligned}
$$

Our learning system found the following definition for the function *irregular*.

$$
\begin{aligned}
&irregular\ p =\\
&\quad if\ (sublists\ 3) \cdot (setExists_1\ (\wedge_3\ (!!0) \cdot (= S)\ (!!1) \cdot (= A)\ (!!2) \cdot (= A)))\ p\\
&\quad then\ \top\\
&\quad else\ \bot.
\end{aligned}
$$

"A protein is irregular if and only if it contains the subsequence SAA (that is, Serine, Alanine, and Alanine) of amino acids ".

5.7 Binary trees

As an example for binary trees, we consider the problem of determining whether a binary tree is balanced or not. Define the following type.

$$SearchTree = BTree\ Char.$$

We want to learn the function *balanced* with the following signature.

$$balanced : SearchTree \rightarrow \Omega.$$

Here are the examples. (The tree from the fourth example is illustrated in Figure 3.)

$$
\begin{aligned}
&balanced\ (BNode\ (BNode\ Null\ 'A'\ Null)\ 'B'\ (BNode\ Null\ 'C'\ Null)) = \top\\
&balanced\ (BNode\ (BNode\ Null\ 'N'\ Null)\ 'R'\ Null) = \top\\
&balanced\ (BNode\ (BNode\ Null\ 'Q'\ Null)\ 'T'\ (BNode\ Null\ 'W'\ Null)) = \top\\
&balanced\ (BNode\ (BNode\ (BNode\ Null\ 'B'\ Null)\ 'D'\ Null)\ 'P'\ (BNode\ Null\ 'T'\ Null)) = \top\\
&balanced\ (BNode\ (BNode\ Null\ 'C'\ (BNode\ Null\ 'L'\ Null))\ 'M'\ (BNode\ Null\ 'W'\ Null)) = \top\\
&balanced\ (BNode\ Null\ 'G'\ (BNode\ Null\ 'N'\ Null)) = \bot\\
&balanced\ (BNode\ Null\ 'P'\ (BNode\ Null\ 'Q'\ (BNode\ Null\ 'Z'\ Null))) = \bot\\
&balanced\ (BNode\ Null\ 'A'\ (BNode\ Null\ 'R'\ (BNode\ Null\ 'S'\ Null))) = \bot.
\end{aligned}
$$

The hypothesis language contains the following transformations.

$$
\begin{aligned}
&(= 'A') : Char \rightarrow \Omega\\
&\qquad\vdots\\
&(= 'Z') : Char \rightarrow \Omega\\
&emptyTree : SearchTree \rightarrow \Omega\\
&root : SearchTree \rightarrow Char\\
&leftTree : SearchTree \rightarrow SearchTree\\
&rightTree : SearchTree \rightarrow SearchTree\\
&nodeSet : SearchTree \rightarrow \{Char\}\\
&setExists_1 : (Char \rightarrow \Omega) \rightarrow \{Char\} \rightarrow \Omega\\
&setExists_2 : (Char \rightarrow \Omega) \rightarrow (Char \rightarrow \Omega) \rightarrow \{Char\} \rightarrow \Omega.
\end{aligned}
$$

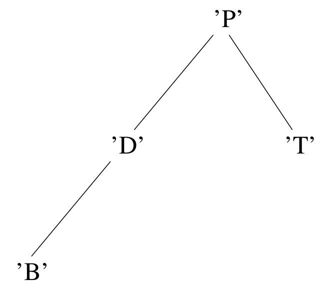

Figure 3: The tree $BNode\ (BNode\ (BNode\ Null\ 'B'\ Null)\ 'D'\ Null)\ 'P'\ (BNode\ Null\ 'T'\ Null)$

Our learning system found the following definition for the function *balanced*.

$$
\begin{array}{l}
balanced\ t = \\
\quad if\ leftTree \cdot emptyTree\ t \\
\quad then\ \bot \\
\quad else\ \top.
\end{array}
$$

"A search tree is not balanced if and only if it has an empty left subtree".

5.8 Trees

To illustrate the use of general trees, we consider trees whose nodes are integers. This leads to the following type synonym.

$$IntegralTree = Tree\ Int.$$

Although the problem below is artificial, it demonstrates how relatively complex predicates can be constructed for trees. We want to learn the following function.

$$interesting : IntegralTree \to \Omega.$$

The examples are as follows. (The tree from the second example is illustrated in Figure 4.)

$$
\begin{array}{l}
interesting\ (Node\ 32\ [Node\ 67\ [], Node\ 42\ [], Node\ 9\ []]) = \top \\
interesting\ (Node\ 89\ [Node\ 6\ [Node\ 8\ [], Node\ 22\ [], Node\ 30\ [], Node\ 31\ [], Node\ 32\ []], \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Node\ 10\ [], Node\ 78\ []]) = \top \\
interesting\ (Node\ 9\ [Node\ 6\ [Node\ 22\ [], Node\ 88\ [], Node\ 45\ [], Node\ 17\ []], Node\ 1\ [], \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Node\ 44\ []]) = \top \\
interesting\ (Node\ 42\ []) = \top \\
interesting\ (Node\ 33\ [Node\ 7\ [Node\ 42\ []]]) = \top \\
interesting\ (Node\ 42\ [Node\ 47\ [], Node\ 21\ [], Node\ 22\ []]) = \bot \\
interesting\ (Node\ 2\ [Node\ 5\ [], Node\ 19\ []]) = \bot \\
interesting\ (Node\ 31\ [Node\ 15\ [], Node\ 17\ [], Node\ 12\ []]) = \bot \\
interesting\ (Node\ 27\ [Node\ 2\ [], Node\ 6\ []]) = \bot \\
interesting\ (Node\ 9\ [Node\ 6\ [Node\ 22\ [], Node\ 88\ [], Node\ 17\ []], Node\ 10\ [], Node\ 78\ []]) = \bot.
\end{array}
$$

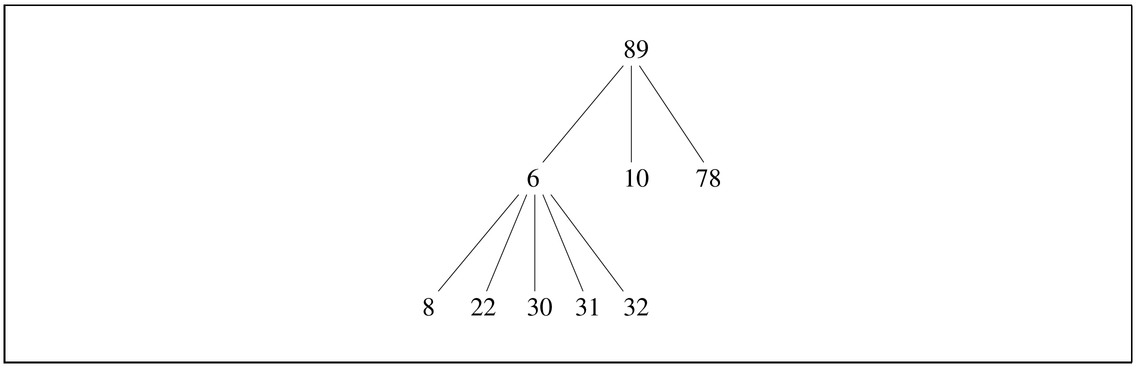

Figure 4: The tree $\mathit{Node}\ 89\ [\mathit{Node}\ 6\ [\mathit{Node}\ 8\ [], \mathit{Node}\ 22\ [], \mathit{Node}\ 30\ [], \mathit{Node}\ 31\ [], \mathit{Node}\ 32\ []], \mathit{Node}\ 10\ [], \mathit{Node}\ 78\ []]$

The hypothesis language contains the following transformations.

$$
\begin{aligned}
&(= 1) : \mathit{Int} \to \Omega \\
&\qquad\vdots \\
&(= 89) : \mathit{Int} \to \Omega \\
&(> 1) : \mathit{Int} \to \Omega \\
&(> 2) : \mathit{Int} \to \Omega \\
&\qquad\vdots \\
&(> 10) : \mathit{Int} \to \Omega \\
&\mathit{root} : \mathit{IntegralTree} \to \mathit{Int} \\
&\mathit{nodeSet} : \mathit{IntegralTree} \to \{\mathit{Int}\} \\
&\mathit{branches} : \mathit{IntegralTree} \to \mathit{List}\ (\mathit{List}\ \mathit{Int}) \\
&\mathit{listToSet} : \mathit{List}\ (\mathit{List}\ \mathit{Int}) \to \{\mathit{List}\ \mathit{Int}\} \\
&\mathit{null} : \mathit{List}\ \mathit{Int} \to \Omega \\
&\mathit{head} : \mathit{List}\ \mathit{Int} \to \mathit{Int} \\
&\mathit{last} : \mathit{List}\ \mathit{Int} \to \mathit{Int} \\
&\mathit{setExists}_1 : (\mathit{Int} \to \Omega) \to \{\mathit{Int}\} \to \Omega \\
&\mathit{setExists}_2 : (\mathit{Int} \to \Omega) \to (\mathit{Int} \to \Omega) \to \{\mathit{Int}\} \to \Omega \\
&\mathit{setExists}_1 : (\mathit{List}\ \mathit{Int} \to \Omega) \to \{\mathit{List}\ \mathit{Int}\} \to \Omega \\
&\mathit{domCard} : (\mathit{Int} \to \Omega) \to \{\mathit{Int}\} \to \mathit{Int} \\
&\wedge_2 : (\mathit{List}\ \mathit{Int} \to \Omega) \to (\mathit{List}\ \mathit{Int} \to \Omega) \to \mathit{List}\ \mathit{Int} \to \Omega \\
&\wedge_3 : (\mathit{List}\ \mathit{Int} \to \Omega) \to (\mathit{List}\ \mathit{Int} \to \Omega) \to (\mathit{List}\ \mathit{Int} \to \Omega) \to \mathit{List}\ \mathit{Int} \to \Omega.
\end{aligned}
$$

Our learning system found the following definition for the function *interesting*.

$$
\begin{aligned}
&\mathit{interesting}\ t = \\
&\quad \mathit{if}\ \mathit{branches} \cdot \mathit{listToSet} \cdot (\mathit{setExists}_1\ (\mathit{last} \cdot (= 42)))\ t \\
&\quad \mathit{then}\ \top \\
&\quad \mathit{else}\ \mathit{if}\ \mathit{nodeSet} \cdot (\mathit{domCard}\ \mathit{top}) \cdot (> 7)\ t \\
&\qquad \mathit{then}\ \top \\
&\qquad \mathit{else}\ \bot.
\end{aligned}
$$

"If an integral tree has a branch which ends in a node with 42, then it is interesting; else if it has more than 7 (distinct) nodes, then it is interesting; else it is not interesting".

5.9 Digraphs

The next application is problem 47 in the book [Bon70, p.229] of Bongard on pattern recognition. This problem is illustrated in Figure 5.

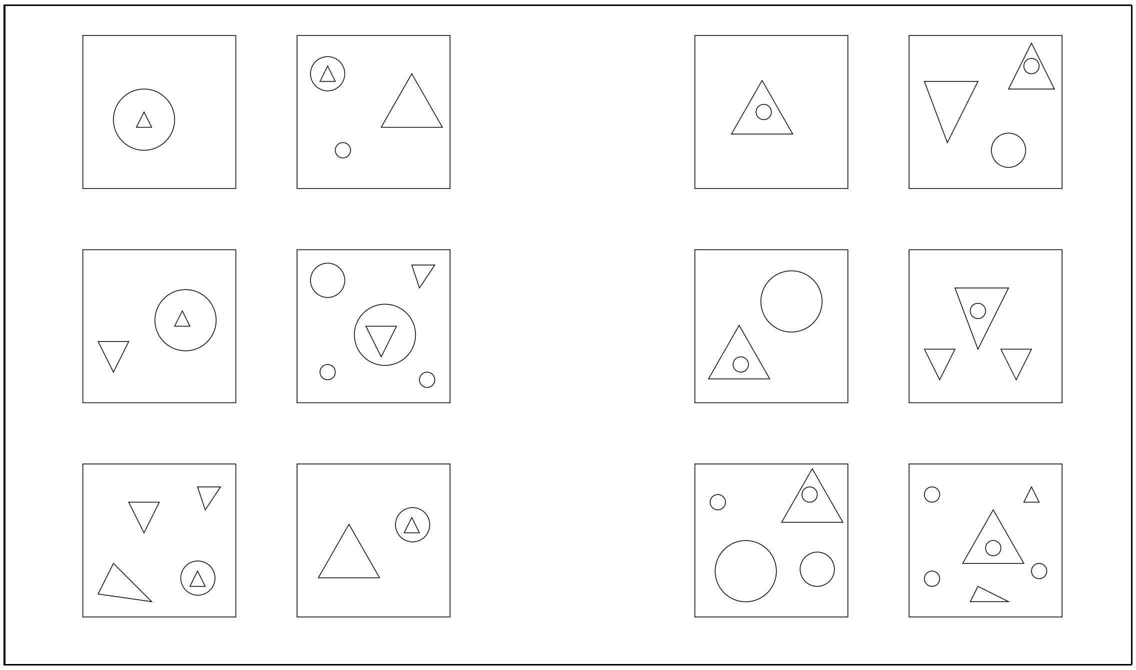

Figure 5: Bongard problem 47: Examples in *Class1* are on the left; those in *Class2* on the right

As usual, we must first decide on a suitable type to represent a Bongard diagram. For this, we have chosen to use a digraph. The geometric figures in the diagram, which can be circles, triangles, or squares, are the nodes of the graph and there is a directed edge from shape s_1 to shape s_2 iff s_2 is located inside s_1.

We introduce the types *Shape* and *Class* as follows.

$$\mathit{Circle}, \mathit{Triangle}, \mathit{Square} : \mathit{Shape}$$
$$\mathit{Class1}, \mathit{Class2} : \mathit{Class}.$$

In this application, there is no information on the edges of the graph, so we use the type *1* as a default for the type of this information and the empty tuple () is used for all these values. The type *Diagram* is defined as follows.

$$\mathit{Diagram} = \mathit{Digraph}\ \mathit{Shape}\ \mathit{1}.$$

Vertices contain information of type *Shape*, while edges contain information of type *1* (which is equivalent to having no information at all). Thus the term

$$(\{(1, \mathit{Circle}), (2, \mathit{Triangle})\}, \{((1, 2), ())\})$$

represents the diagram with a circle and a triangle, and where the triangle is inside the circle.

We want to learn the definition of the function *class* having signature as follows.

$$\mathit{class} : \mathit{Diagram} \rightarrow \mathit{Class}.$$

Here are the examples.

$$
\begin{aligned}
&\mathit{class}\ (\{(1, \mathit{Circle}), (2, \mathit{Triangle})\}, \{((1,2),())\}) = \mathit{Class1}\\
&\mathit{class}\ (\{(1, \mathit{Circle}), (2, \mathit{Triangle}), (3, \mathit{Triangle}), (4, \mathit{Circle})\}, \{((1,2),())\}) = \mathit{Class1}\\
&\mathit{class}\ (\{(1, \mathit{Triangle}), (2, \mathit{Circle}), (3, \mathit{Triangle})\}, \{((2,3),())\}) = \mathit{Class1}\\
&\mathit{class}\ (\{(1, \mathit{Circle}), (2, \mathit{Triangle}), (3, \mathit{Circle}), (4, \mathit{Triangle}), (5, \mathit{Circle}), (6, \mathit{Circle})\},\\
&\hspace{20em} \{((3,4),())\}) = \mathit{Class1}\\
&\mathit{class}\ (\{(1, \mathit{Triangle}), (2, \mathit{Triangle}), (3, \mathit{Triangle}), (4, \mathit{Circle}), (5, \mathit{Triangle})\},\\
&\hspace{20em} \{((4,5),())\}) = \mathit{Class1}\\
&\mathit{class}\ (\{(1, \mathit{Triangle}), (2, \mathit{Circle}), (3, \mathit{Triangle})\}, \{((2,3),())\}) = \mathit{Class1}\\
&\mathit{class}\ (\{(1, \mathit{Triangle}), (2, \mathit{Circle})\}, \{((1,2),())\}) = \mathit{Class2}\\
&\mathit{class}\ (\{(1, \mathit{Triangle}), (2, \mathit{Triangle}), (3, \mathit{Circle}), (4, \mathit{Circle})\}, \{((2,3),())\}) = \mathit{Class2}\\
&\mathit{class}\ (\{(1, \mathit{Circle}), (2, \mathit{Triangle}), (3, \mathit{Circle})\}, \{((2,3),())\}) = \mathit{Class2}\\
&\mathit{class}\ (\{(1, \mathit{Triangle}), (2, \mathit{Circle}), (3, \mathit{Triangle}), (4, \mathit{Triangle})\}, \{((1,2),())\}) = \mathit{Class2}\\
&\mathit{class}\ (\{(1, \mathit{Circle}), (2, \mathit{Triangle}), (3, \mathit{Circle}), (4, \mathit{Circle}), (5, \mathit{Circle})\}, \{((2,3),())\}) =\\
&\hspace{28em} \mathit{Class2}\\
&\mathit{class}\ (\{(1, \mathit{Circle}), (2, \mathit{Triangle}), (3, \mathit{Triangle}), (4, \mathit{Circle}), (5, \mathit{Circle}), (6, \mathit{Triangle}),\\
&\hspace{16em} (7, \mathit{Circle})\}, \{((3,4),())\}) = \mathit{Class2}.
\end{aligned}
$$

The hypothesis language contains the following transformations.

$$
\begin{aligned}
&(= \mathit{Circle}) : \mathit{Shape} \to \Omega\\
&(= \mathit{Triangle}) : \mathit{Shape} \to \Omega\\
&(= \mathit{Square}) : \mathit{Shape} \to \Omega\\
&\mathit{vertices} : \mathit{Diagram} \to \{\mathit{DiVertex\ Shape\ 1}\}\\
&\mathit{edges} : \mathit{Diagram} \to \{\mathit{DiEdge\ Shape\ 1}\}\\
&\mathit{vertex} : \mathit{DiVertex\ Shape\ 1} \to \mathit{Shape}\\
&\mathit{edge} : \mathit{DiEdge\ Shape\ 1} \to \mathit{1}\\
&\mathit{connects} : \mathit{DiEdge\ Shape\ 1} \to \mathit{DiVertex\ Shape\ 1} \times \mathit{DiVertex\ Shape\ 1}\\
&\mathit{projDiVertex1} : \mathit{DiVertex\ Shape\ 1} \times \mathit{DiVertex\ Shape\ 1} \to \mathit{DiVertex\ Shape\ 1}\\
&\mathit{projDiVertex2} : \mathit{DiVertex\ Shape\ 1} \times \mathit{DiVertex\ Shape\ 1} \to \mathit{DiVertex\ Shape\ 1}\\
&\mathit{setExists}_1 : (\mathit{DiVertex\ Shape\ 1} \to \Omega) \to \{\mathit{DiVertex\ Shape\ 1}\} \to \Omega\\
&\mathit{setExists}_1 : (\mathit{DiEdge\ Shape\ 1} \to \Omega) \to \{\mathit{DiEdge\ Shape\ 1}\} \to \Omega\\
&\wedge_2 : (\mathit{DiVertex\ Shape\ 1} \to \Omega) \to (\mathit{DiVertex\ Shape\ 1} \to \Omega) \to \mathit{DiVertex\ Shape\ 1} \to \Omega.
\end{aligned}
$$

Here *projDiVertex1* projects onto the first component and *projDiVertex2* projects onto the second.

Our learning system found the following definition for the function *class*.

$$
\begin{aligned}
&\mathit{class}\ d =\\
&\quad \mathit{if}\ \ \mathit{edges} \cdot (\mathit{setExists}_1\ (\mathit{connects} \cdot \mathit{projDiVertex2} \cdot \mathit{vertex} \cdot (= \mathit{Circle})))\ d\\
&\quad \mathit{then}\ \mathit{Class2}\\
&\quad \mathit{else}\ \mathit{Class1}.
\end{aligned}
$$

“A diagram is in class 2 if it contains a circle inside something; otherwise, it is in class 1”.

5.10 Graphs

The next application involves learning a theory to predict whether a chemical molecule has a certain property. In spite of appearances, this application is quite artificial – we study the mutagenesis problem, a similar, but real, application later.

As explained earlier, 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.

$$Br, C, Cl, F, H, I, N, O, S : Element.$$

We also make the following type synonyms.

$$\begin{aligned}
&AtomType = Nat\\
&Charge = Float\\
&Atom = Element \times AtomType \times Charge\\
&Bond = Nat.
\end{aligned}$$

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.

$$Molecule = Graph\ Atom\ Bond.$$

We want to learn the definition of the function *active* having the following signature.

$$active : Molecule \to \Omega.$$

The examples are as follows.

$$\begin{aligned}
&active\ (\{(1,(C,22,-0.117)),(2,(O,45,-0.388)),(3,(C,22,-0.117)),\\
&\qquad (4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),\\
&\qquad (7,(O,40,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142)),\\
&\qquad (10,(H,3,0.142)),(11,(C,27,-0.087)),(12,(C,27,0.013))\},\\
&\quad \{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),(\langle 3,4\rangle,7),\\
&\qquad (\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 4,11\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,2),\\
&\qquad (\langle 6,8\rangle,2),(\langle 11,12\rangle,7)\}) = \top\\
&active\ (\{(1,(C,22,-0.117)),(2,(C,22,-0.117)),(3,(C,22,-0.117)),\\
&\qquad (4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),\\
&\qquad (7,(O,40,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142)),\\
&\qquad (10,(H,3,0.142)),(11,(C,27,-0.087)),(12,(C,27,0.013)),\\
&\qquad (13,(C,22,-0.117))\},\\
&\quad \{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&\qquad (\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 4,11\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,2),\\
&\qquad (\langle 6,8\rangle,2),(\langle 11,12\rangle,7),(\langle 12,13\rangle,7)\}) = \top\\
&active\ (\{(1,(H,3,0.142)),(2,(C,22,-0.117)),(3,(C,22,-0.117)),\\
&\qquad (4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),\\
&\qquad (7,(O,40,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142))\},\\
&\quad \{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&\qquad (\langle 3,4\rangle,7),(\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,2),\\
&\qquad (\langle 6,8\rangle,2)\}) = \top\\
&active\ (\{(1,(C,22,-0.117)),(2,(C,22,-0.117)),(3,(C,22,-0.117)),\\
&\qquad (4,(C,195,-0.087)),(5,(H,3,0.143)),(6,(N,38,0.812)),\\
&\qquad (7,(O,40,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142)),
\end{aligned}$$

$$
\begin{aligned}
&\quad (10,(H,3,0.142)),(11,(C,27,-0.087)),(12,(C,27,0.013)),\\
&\quad (13,(C,22,-0.117))\},\\
&\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&\quad (\langle 3,4\rangle,7),(\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 4,11\rangle,7),(\langle 5,6\rangle,7),\\
&\quad (\langle 6,7\rangle,2),(\langle 6,8\rangle,2),(\langle 11,12\rangle,7),(\langle 12,13\rangle,7)\}) = \top\\
\mathit{active}\ &(\{(1,(H,3,0.142)),(2,(C,22,-0.117)),(3,(O,45,-0.338)),\\
&\quad (4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),\\
&\quad (7,(O,40,-0.388)),(8,(O,40,-0.388))\},\\
&\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&\quad (\langle 3,4\rangle,7),(\langle 4,5\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,2),(\langle 6,8\rangle,2)\}) = \top\\
\mathit{active}\ &(\{(1,(C,22,-0.117)),(2,(O,45,-0.388)),(3,(C,22,-0.117)),\\
&\quad (4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),\\
&\quad (7,(O,45,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142)),\\
&\quad (10,(H,3,0.142)),(11,(C,27,-0.087)),(12,(C,27,0.013))\},\\
&\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&\quad (\langle 3,4\rangle,7),(\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 4,11\rangle,7),(\langle 5,6\rangle,7),\\
&\quad (\langle 6,7\rangle,2),(\langle 6,8\rangle,2),(\langle 11,12\rangle,7)\}) = \bot\\
\mathit{active}\ &(\{(1,(H,3,0.142)),(2,(C,22,-0.117)),(3,(O,45,-0.338)),\\
&\quad (4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),\\
&\quad (7,(O,40,-0.388)),(8,(O,40,-0.388))\},\\
&\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&\quad (\langle 3,4\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,2)\}) = \bot\\
\mathit{active}\ &(\{(1,(H,3,0.142)),(2,(C,22,-0.117)),(3,(C,22,-0.117)),\\
&\quad (4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),\\
&\quad (7,(O,45,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142))\},\\
&\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,8\rangle,1),(\langle 3,9\rangle,1),\\
&\quad (\langle 4,5\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,2),(\langle 6,8\rangle,3)\}) = \bot\\
\mathit{active}\ &(\{(1,(C,22,-0.117)),(2,(C,22,-0.117)),(3,(C,22,-0.117)),\\
&\quad (4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,35,0.812)),\\
&\quad (7,(O,40,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142)),\\
&\quad (10,(H,3,0.142)),(11,(C,27,-0.087)),(12,(C,27,0.013)),\\
&\quad (13,(C,22,-0.117))\},\\
&\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 3,4\rangle,7),\\
&\quad (\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 4,11\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,2),\\
&\quad (\langle 6,8\rangle,2),(\langle 11,12\rangle,7),(\langle 12,13\rangle,7)\}) = \bot\\
\mathit{active}\ &(\{(1,(H,3,0.142)),(2,(C,22,-0.117)),(3,(C,22,-0.117)),\\
&\quad (4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),\\
&\quad (7,(O,40,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142))\},\\
&\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&\quad (\langle 3,4\rangle,7),(\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,3),\\
&\quad (\langle 6,8\rangle,2)\}) = \bot.
\end{aligned}
$$

The hypothesis language contains the following transformations.

$(= \mathit{Br}) : \mathit{Element} \to \Omega$

$$
\begin{aligned}
&\qquad\vdots\\
&(=S) : \mathit{Element} \to \Omega\\
&(=3) : \mathit{AtomType} \to \Omega\\
&\qquad\vdots\\
&(=195) : \mathit{AtomType} \to \Omega\\
&(\leq -0.117) : \mathit{Charge} \to \Omega\\
&(\leq -0.087) : \mathit{Charge} \to \Omega\\
&(\leq 0.013) : \mathit{Charge} \to \Omega\\
&(\leq 0.142) : \mathit{Charge} \to \Omega\\
&(\geq -0.117) : \mathit{Charge} \to \Omega\\
&(\geq -0.087) : \mathit{Charge} \to \Omega\\
&(\geq 0.013) : \mathit{Charge} \to \Omega\\
&(\geq 0.142) : \mathit{Charge} \to \Omega\\
&(=1) : \mathit{Bond} \to \Omega\\
&(=2) : \mathit{Bond} \to \Omega\\
&(=3) : \mathit{Bond} \to \Omega\\
&(=7) : \mathit{Bond} \to \Omega\\
&(>0) : \mathit{Nat} \to \Omega\\
&(>1) : \mathit{Nat} \to \Omega\\
&(>2) : \mathit{Nat} \to \Omega\\
&\mathit{projElement} : \mathit{Atom} \to \mathit{Element}\\
&\mathit{projAtomType} : \mathit{Atom} \to \mathit{AtomType}\\
&\mathit{projCharge} : \mathit{Atom} \to \mathit{Charge}\\
&\mathit{vertices} : \mathit{Molecule} \to \{\mathit{Vertex\ Atom\ Bond}\}\\
&\mathit{edges} : \mathit{Molecule} \to \{\mathit{Edge\ Atom\ Bond}\}\\
&\mathit{vertex} : \mathit{Vertex\ Atom\ Bond} \to \mathit{Atom}\\
&\mathit{connects} : \mathit{Edge\ Atom\ Bond} \to (\mathit{Vertex\ Atom\ Bond} \to \mathit{Nat})\\
&\mathit{edge} : \mathit{Edge\ Atom\ Bond} \to \mathit{Bond}\\
&\mathit{domCard} : (\mathit{Vertex\ Atom\ Bond} \to \Omega) \to \{\mathit{Vertex\ Atom\ Bond}\} \to \mathit{Nat}\\
&\mathit{domCard} : (\mathit{Edge\ Atom\ Bond} \to \Omega) \to \{\mathit{Edge\ Atom\ Bond}\} \to \mathit{Nat}\\
&\mathit{setExists}_1 : (\mathit{Molecule} \to \Omega) \to \{\mathit{Molecule}\} \to \Omega\\
&(\mathit{subgraphs}\ 3) : \mathit{Molecule} \to \{\mathit{Molecule}\}\\
&\mathit{msetExists}_2 : (\mathit{Vertex\ Atom\ Bond} \to \Omega) \to (\mathit{Vertex\ Atom\ Bond} \to \Omega) \to\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Vertex\ Atom\ Bond} \to \mathit{Nat}) \to \Omega\\
&\wedge_2 : (\mathit{Charge} \to \Omega) \to (\mathit{Charge} \to \Omega) \to \mathit{Charge} \to \Omega\\
&\wedge_2 : (\mathit{Atom} \to \Omega) \to (\mathit{Atom} \to \Omega) \to \mathit{Atom} \to \Omega\\
&\wedge_3 : (\mathit{Atom} \to \Omega) \to (\mathit{Atom} \to \Omega) \to (\mathit{Atom} \to \Omega) \to \mathit{Atom} \to \Omega\\
&\wedge_2 : (\mathit{Edge\ Atom\ Bond} \to \Omega) \to (\mathit{Edge\ Atom\ Bond} \to \Omega) \to \mathit{Edge\ Atom\ Bond} \to \Omega\\
&\wedge_2 : (\mathit{Molecule} \to \Omega) \to (\mathit{Molecule} \to \Omega) \to \mathit{Molecule} \to \Omega.
\end{aligned}
$$

Our learning system found the following definition for the function *active*.

$$\begin{aligned}
&active\ m = \\
&\quad if\ (subgraphs\ 3) \cdot (setExists_1\ (\wedge_2 \\
&\qquad\qquad (vertices \cdot (domCard\ (vertex \cdot projAtomType \cdot (= 38))) \cdot (> 0)) \\
&\qquad\qquad (edges \cdot (domCard\ (\wedge_2 \\
&\qquad\qquad\quad (connects \cdot (msetExists_2\ (vertex \cdot projAtomType \cdot (= 40))\ top)) \\
&\qquad\qquad\quad (edge \cdot (= 2)))) \cdot (> 1))))\ m \\
&\quad then\ \top \\
&\quad else\ \bot.
\end{aligned}$$

“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”.

6 Predicate Construction

In this section, we describe the algorithm for constructing predicates. To motivate the ideas, we start with an extended example.

Example. Consider the keys application in Subsection 5.2 again. Recall that the transformations for this application were as follows.

$$\begin{aligned}
&(= Abloy) : Make \rightarrow \Omega \\
&(= Chubb) : Make \rightarrow \Omega \\
&(= Rubo) : Make \rightarrow \Omega \\
&(= Yale) : Make \rightarrow \Omega \\
&(= 2) : NumProngs \rightarrow \Omega \\
&(= 3) : NumProngs \rightarrow \Omega \\
&(= 4) : NumProngs \rightarrow \Omega \\
&(= 5) : NumProngs \rightarrow \Omega \\
&(= 6) : NumProngs \rightarrow \Omega \\
&(= Short) : Length \rightarrow \Omega \\
&(= Medium) : Length \rightarrow \Omega \\
&(= Long) : Length \rightarrow \Omega \\
&(= Narrow) : Width \rightarrow \Omega \\
&(= Normal) : Width \rightarrow \Omega \\
&(= Broad) : Width \rightarrow \Omega \\
&prMake : Key \rightarrow Make \\
&prNumProngs : Key \rightarrow NumProngs \\
&prLength : Key \rightarrow Length \\
&prWidth : Key \rightarrow Width \\
&setExists_1 : (Key \rightarrow \Omega) \rightarrow Bunch \rightarrow \Omega \\
&\wedge_2 : (Key \rightarrow \Omega) \rightarrow (Key \rightarrow \Omega) \rightarrow Key \rightarrow \Omega \\
&\wedge_3 : (Key \rightarrow \Omega) \rightarrow (Key \rightarrow \Omega) \rightarrow (Key \rightarrow \Omega) \rightarrow Key \rightarrow \Omega \\
&\wedge_4 : (Key \rightarrow \Omega) \rightarrow (Key \rightarrow \Omega) \rightarrow (Key \rightarrow \Omega) \rightarrow (Key \rightarrow \Omega) \rightarrow Key \rightarrow \Omega.
\end{aligned}$$

(Here *projMake* is abbreviated to *prMake*, and so on.) Starting from the weakest predicate *top* (having type $Bunch \to \Omega$), we show how to systematically construct predicates using a system of rewrites. First, here are the rewrites for this application.

$$
\begin{aligned}
&top \rightarrowtail setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot top))\\
&top \rightarrowtail (=Abloy)\\
&top \rightarrowtail (=Chubb)\\
&top \rightarrowtail (=Rubo)\\
&top \rightarrowtail (=Yale)\\
&top \rightarrowtail (=2)\\
&top \rightarrowtail (=3)\\
&top \rightarrowtail (=4)\\
&top \rightarrowtail (=5)\\
&top \rightarrowtail (=6)\\
&top \rightarrowtail (=Short)\\
&top \rightarrowtail (=Medium)\\
&top \rightarrowtail (=Long)\\
&top \rightarrowtail (=Narrow)\\
&top \rightarrowtail (=Normal)\\
&top \rightarrowtail (=Broad).
\end{aligned}
$$

Each rewrite has the form $r \rightarrowtail s$, where r and s are predicates (of the same type). When r occurs as a redex in a predicate p and the rewrite $r \rightarrowtail s$ 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 $Bunch \to \Omega$, while the first occurrence of *top* in the body has type $Make \to \Omega$.

One of the most delicate aspects of predicate construction 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 $prMake \cdot top, \ldots, prWidth \cdot top$. Note that each of these predicates is equivalent to *top*, so that they do not actually constrain the components. As the enumeration of predicates progresses, the *top*'s in the conjunction are replaced by stronger predicates. Note that a predicate such as

$$\wedge_4\ (prMake \cdot (=Abloy))\ (prNumProngs \cdot top)\ (prLength \cdot (=Medium))\ (prWidth \cdot top)$$

can be simplified to the equivalent predicate

$$\wedge_2\ (prMake \cdot (=Abloy))\ (prLength \cdot (=Medium)).$$

The procedure for constructing predicates is as follows. Starting with the predicate *top* (of type $Bunch \to \Omega$), all rewrites whose head matches *top* are applied. There is only one of these,

$$top \rightarrowtail setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot top)),$$

which gives the new predicate

$$setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot top)).$$

There are four redexes in this predicate, each having the form *top*. When the rewrites whose heads match these redexes are applied, the following predicates are obtained.

$$
\begin{array}{l}
setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot top)\ (prLength \cdot top)\ prWidth \cdot top))\\
setExists_1\ (\wedge_4\ (prMake \cdot (= Chubb))\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot top))\\
setExists_1\ (\wedge_4\ (prMake \cdot (= Rubo))\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot top))\\
setExists_1\ (\wedge_4\ (prMake \cdot (= Yale))\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot top))\\
setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot (= 2))\ (prLength \cdot top)\ (prWidth \cdot top))\\
setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot (= 3))\ (prLength \cdot top)\ (prWidth \cdot top))\\
\qquad\qquad\vdots\\
setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot (= Narrow)))\\
setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot (= Normal)))\\
setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot (= Broad))).
\end{array}
$$

Taking the first of these children,

$$setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot top)),$$

and applying the rewrites at each available redex, one obtains

$$
\begin{array}{l}
setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot (= 2))\ (prLength \cdot top)\ (prWidth \cdot top))\\
setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot (= 3))\ (prLength \cdot top)\ (prWidth \cdot top))\\
\qquad\qquad\vdots\\
setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot (= Narrow)))\\
setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot (= Normal)))\\
setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot (= Broad))).
\end{array}
$$

This process continues, producing successively stronger predicates. This completes the example.

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, called standard predicates, under consideration. A standard predicate is defined by induction on the number of transformations it contains.

Definition. A *standard predicate* is a term of the form

$$(f_1\ b_{1,1} \ldots b_{1,k_1}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n}),$$

where f_i is a transformation of rank k_i $(i = 1, \ldots, n)$, the target of f_n is Ω, b_{i,j_i} is a standard predicate $(i = 1, \ldots, n,\ \ j_i = 1, \ldots, k_i)$, $k_i \geq 0$ $(i = 1, \ldots, n)$ and $n \geq 1$.

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. Let $\mathcal{B}$ denote the theory consisting of the definitions of the transformations (and associated functions). This theory is usually called the *background theory*.

Definition. The relation $\Leftarrow$ on the class of standard predicates on the same type is defined by $p \Leftarrow q$ if $\forall x.((p\ x) \longleftarrow (q\ x))$ is a logical consequence of $\mathcal{B}$.

(Actually, this definition is somewhat informal – the precise definition is given in the Appendix.) 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 that implies the parent, then we need to ensure that rewrites respect the implication preorder. More precisely, if $p \rightarrowtail q$ is a rewrite, then $p \Leftarrow q$ should hold. Clearly all the rewrites in the previous example satisfy this property.

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 $(domCard\ p) \cdot (< 42)$ and suppose there is a rewrite $p \rightarrowtail q$ (where q implies p, of course). If we apply the rewrite, then we obtain the child predicate $(domCard\ q) \cdot (< 42)$. However, it is not true that $(domCard\ q) \cdot (< 42)$ implies $(domCard\ p) \cdot (< 42)$.

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 technical details of how this is achieved require too much development to be included here, but can be found in [Llo01b]. In summary, one defines the *monotone positions* in a predicate at which redexes can be present and restricts the application of the rewrites to these redexes. These monotone positions are identified by declaring skeletons and their monotone arguments. Skeletons are something like predicates, but with each argument position occupied by a unique variable. In fact, while they do occur, redexes not appearing in a monotone position are rather rare – for example, there are none in this paper (apart from the previous example, of course).

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). Even the simple keys application suffers from this problem, as there are two paths to any conjunction of two predicates, and more for bigger conjunctions. 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. This tactic essentially turns the search space back into a tree.

The last complication comes from the equivalence of syntactically distinct predicates. To see the problem, suppose that for the keys application we decided to restrict the number of conjuncts to two. This can be enforced by the following rewrites.

$$
\begin{aligned}
&top \rightarrowtail setExists_1\ (\wedge_2\ top\ top)\\
&top \rightarrowtail prMake \cdot (= Abloy)\\
&top \rightarrowtail prMake \cdot (= Chubb)\\
&\qquad\qquad \vdots\\
&top \rightarrowtail prWidth \cdot (= Normal)\\
&top \rightarrowtail prWidth \cdot (= Broad).
\end{aligned}
$$

If a tuple had a large number of components, it could be natural to use rewrites of this form. Now, for this collection of rewrites, the top level of the search space contains the predicates

$$
\begin{aligned}
&setExists_1\ (\wedge_2\ (prMake \cdot (= Abloy))\ top)\\
&setExists_1\ (\wedge_2\ (prMake \cdot (= Chubb))\ top)\\
&\qquad\qquad \vdots\\
&setExists_1\ (\wedge_2\ (prWidth \cdot (= Normal))\ top)\\
&setExists_1\ (\wedge_2\ (prWidth \cdot (= Broad))\ top)\\
&setExists_1\ (\wedge_2\ top\ (prMake \cdot (= Abloy)))\\
&setExists_1\ (\wedge_2\ top\ (prMake \cdot (= Chubb)))\\
&\qquad\qquad \vdots
\end{aligned}
$$

$$setExists_1\ (\wedge_2\ top\ (prWidth \cdot (= Normal)))$$
$$setExists_1(\wedge_2\ top\ (prWidth \cdot (= Broad)))$$

and, at the next level, predicates such as

$$setExists_1\ (\wedge_2\ (prMake \cdot (= Abloy))\ (prWidth \cdot (= Broad)))$$
$$setExists_1\ (\wedge_2\ (prWidth \cdot (= Broad))\ (prMake \cdot (= Abloy))).$$

Now the problem is apparent: these last two predicates are equivalent since the order of arguments for $\wedge_2$ is immaterial. The problem is even worse with $\wedge_3$, which has 3! predicates in each equivalence class. Similar difficulties occur with $setExists_2$. The obvious solution is to define some total order on predicates and then ensure the arguments to $\wedge_2$, $\wedge_3$, $setExists_2$ and similar predicates, are increasingly ordered according to this order. This will cut out one of the two equivalent predicates just given.

The technical details of this are outlined in the Appendix and given in detail in [Llo01b]. In summary, a total order is first given for the transformations and this total order is then lifted to the predicates. One can then define a restricted class of predicates, called the *regular* predicates, whose arguments are ordered as indicated above and also have the property that, for each (standard) predicate p, there is a regular predicate q such that $p = q$ is a logical consequence of $\mathcal{B}$. Thus nothing is lost by restricting attention to regular predicates. There is also an algorithm that, given a predicate p, constructs a regular predicate q such that $p = q$ is a logical consequence of $\mathcal{B}$. This desire to work with regular predicates also explains why we use $\wedge_2$, $\wedge_3$, $\wedge_4$, and so on, rather than simply using combinations of binary conjunction – it is technically easier to ensure that $\wedge_2$, $\wedge_3$, $\wedge_4$, and so on, are regular. Furthermore, explicitly distinguishing the conjunctions of different arities gives greater control over which predicates are constructed.

For the previous set of rewrites, there is another solution that avoids the problem of generating syntactically distinct predicates that are equivalent. Replace the previous set by the following.

$$top \rightarrowtail setExists_1\ (\wedge_2\ (prMake \cdot top)\ (prNumProngs \cdot top))$$
$$\vdots$$
$$top \rightarrowtail setExists_1\ (\wedge_2\ (prLength \cdot top)\ (prWidth \cdot top))$$
$$top \rightarrowtail (= Abloy)$$
$$\vdots$$
$$top \rightarrowtail (= Broad).$$

This set of rewrites results in the same set of (non-equivalent) predicates as for the previous set, but avoids the problem of constructing syntactically distinct predicates that are equivalent. Our approach to predicate construction provides the flexibility to tailor rewrite sets in this way, but it is not a complete solution to this problem. For example, in Section 8, the first rewrite set for the Musk problem suffers from this problem, but was nonetheless chosen because it is well-suited to a parallel implementation.

The above considerations are summarised in the algorithm for the function *Enumerate*, which is given in Figure 6 below. The input is a rewrite relation $\rightarrowtail$ that holds for regular predicates and respects the implication preorder (that is, $\rightarrowtail$ is a regular, implication-consistent relation) and a set of skeletons with their monotone arguments for recognising monotone positions. The function *Positions* computes monotone positions at which redexes can appear. The function *Regularise* computes a regular predicate that is equivalent to the argument given to the function (by reordering certain subterms of the argument predicate, if necessary). The predicate *top* in the initial open list and seen set is implicitly labelled by its type to avoid ambiguity. If $\rightarrowtail$ is finite (that is, there are finitely many rewrites – the usual case), then each predicate has finitely many children.

We conclude this section with another, more complicated, example.

function $Enumerate(\rightarrowtail, \mathcal{S})$ **returns** a set of regular predicates;
input: $\rightarrowtail$, a regular, implication-consistent relation;
$\mathcal{S}$, a set of skeletons together with their monotone arguments;

$regular := \{\}$;
$openList := [top]$;
$seenSet := \{top\}$;
while $openList \neq []$ **do**
  $p := head(openList)$;
  $openList := tail(openList)$;
  $regular := regular \cup \{p\}$;
  for each r such that $r \in Positions(p, \mathcal{S})$ and $r \rightarrowtail s$, for some s, **do**
    $q := Regularise(p[r/s])$;
    if $q \notin seenSet$
    then
      $seenSet := seenSet \cup \{q\}$;
      $openList := openList \mathbin{++} [q]$;
return $regular$;

Figure 6: Algorithm for enumerating regular predicates

Example. Consider the chemical property application in Subsection 5.10 again. Recall that the transformations for this application were as follows.

$(= Br) : Element \to \Omega$

$\vdots$

$(= S) : Element \to \Omega$
$(= 3) : AtomType \to \Omega$

$\vdots$

$(= 195) : AtomType \to \Omega$
$(\leq -0.117) : Charge \to \Omega$
$(\leq -0.087) : Charge \to \Omega$
$(\leq 0.013) : Charge \to \Omega$
$(\leq 0.142) : Charge \to \Omega$
$(\geq -0.117) : Charge \to \Omega$
$(\geq -0.087) : Charge \to \Omega$
$(\geq 0.013) : Charge \to \Omega$
$(\geq 0.142) : Charge \to \Omega$
$(= 1) : Bond \to \Omega$
$(= 2) : Bond \to \Omega$
$(= 3) : Bond \to \Omega$
$(= 7) : Bond \to \Omega$

$$
\begin{aligned}
&(> 0) : \mathit{Nat} \rightarrow \Omega \\
&(> 1) : \mathit{Nat} \rightarrow \Omega \\
&(> 2) : \mathit{Nat} \rightarrow \Omega \\
&\mathit{projElement} : \mathit{Atom} \rightarrow \mathit{Element} \\
&\mathit{projAtomType} : \mathit{Atom} \rightarrow \mathit{AtomType} \\
&\mathit{projCharge} : \mathit{Atom} \rightarrow \mathit{Charge} \\
&\mathit{vertices} : \mathit{Molecule} \rightarrow \{\mathit{Vertex}\ \mathit{Atom}\ \mathit{Bond}\} \\
&\mathit{edges} : \mathit{Molecule} \rightarrow \{\mathit{Edge}\ \mathit{Atom}\ \mathit{Bond}\} \\
&\mathit{vertex} : \mathit{Vertex}\ \mathit{Atom}\ \mathit{Bond} \rightarrow \mathit{Atom} \\
&\mathit{connects} : \mathit{Edge}\ \mathit{Atom}\ \mathit{Bond} \rightarrow (\mathit{Vertex}\ \mathit{Atom}\ \mathit{Bond} \rightarrow \mathit{Nat}) \\
&\mathit{edge} : \mathit{Edge}\ \mathit{Atom}\ \mathit{Bond} \rightarrow \mathit{Bond} \\
&\mathit{domCard} : (\mathit{Vertex}\ \mathit{Atom}\ \mathit{Bond} \rightarrow \Omega) \rightarrow \{\mathit{Vertex}\ \mathit{Atom}\ \mathit{Bond}\} \rightarrow \mathit{Nat} \\
&\mathit{domCard} : (\mathit{Edge}\ \mathit{Atom}\ \mathit{Bond} \rightarrow \Omega) \rightarrow \{\mathit{Edge}\ \mathit{Atom}\ \mathit{Bond}\} \rightarrow \mathit{Nat} \\
&\mathit{setExists}_1 : (\mathit{Molecule} \rightarrow \Omega) \rightarrow \{\mathit{Molecule}\} \rightarrow \Omega \\
&(\mathit{subgraphs}\ 3) : \mathit{Molecule} \rightarrow \{\mathit{Molecule}\} \\
&\mathit{msetExists}_2 : (\mathit{Vertex}\ \mathit{Atom}\ \mathit{Bond} \rightarrow \Omega) \rightarrow (\mathit{Vertex}\ \mathit{Atom}\ \mathit{Bond} \rightarrow \Omega) \rightarrow \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Vertex}\ \mathit{Atom}\ \mathit{Bond} \rightarrow \mathit{Nat}) \rightarrow \Omega \\
&\wedge_2 : (\mathit{Charge} \rightarrow \Omega) \rightarrow (\mathit{Charge} \rightarrow \Omega) \rightarrow \mathit{Charge} \rightarrow \Omega \\
&\wedge_2 : (\mathit{Atom} \rightarrow \Omega) \rightarrow (\mathit{Atom} \rightarrow \Omega) \rightarrow \mathit{Atom} \rightarrow \Omega \\
&\wedge_3 : (\mathit{Atom} \rightarrow \Omega) \rightarrow (\mathit{Atom} \rightarrow \Omega) \rightarrow (\mathit{Atom} \rightarrow \Omega) \rightarrow \mathit{Atom} \rightarrow \Omega \\
&\wedge_2 : (\mathit{Edge}\ \mathit{Atom}\ \mathit{Bond} \rightarrow \Omega) \rightarrow (\mathit{Edge}\ \mathit{Atom}\ \mathit{Bond} \rightarrow \Omega) \rightarrow \mathit{Edge}\ \mathit{Atom}\ \mathit{Bond} \rightarrow \Omega \\
&\wedge_2 : (\mathit{Molecule} \rightarrow \Omega) \rightarrow (\mathit{Molecule} \rightarrow \Omega) \rightarrow \mathit{Molecule} \rightarrow \Omega.
\end{aligned}
$$

What is a suitable set of rewrites for this application? This, of course, depends to a great extent on the concepts that are expected to be found in the data. In this case, we look for a connected subgraph containing three atoms having some structure that will distinguish the two cases. (We make this choice because we made up the data and know what to look for! In the usual case of real world data and unknown hypothesis language, we must bring to bear whatever knowledge we can to tease out hidden structure and, in the worst case, systematically try different hypothesis languages until something interesting turns up.) This suggests a top-level rewrite of the form

$$\mathit{top} \rightarrowtail (\mathit{subgraphs}\ 3) \cdot (\mathit{setExists}_1\ \mathit{top}).$$

However, there is a problem with this rewrite. The *top* in the body has type $\mathit{Molecule} \rightarrow \Omega$ (the same as the type of the *top* in the head), so we would end up making further calls on (*subgraphs* 3) when, of course, only one such call makes sense.

Fortunately, the rewrite view of predicate construction is flexible enough to avoid this problem. For example, we could replace the above rewrite by

$$
\begin{aligned}
&\mathit{top} \rightarrowtail (\mathit{subgraphs}\ 3) \cdot (\mathit{setExists}_1\ (\mathit{vertices} \cdot \mathit{top})) \\
&\mathit{top} \rightarrowtail (\mathit{subgraphs}\ 3) \cdot (\mathit{setExists}_1\ (\mathit{edges} \cdot \mathit{top})).
\end{aligned}
$$

Now the *top*s in the bodies have types which avoid the loop. With these two rewrites, we are looking for a property with conditions on either the atoms or the bonds in the subgraph. We can combine these by using the rewrite

$$\mathit{top} \rightarrowtail (\mathit{subgraphs}\ 3) \cdot (\mathit{setExists}_1\ (\wedge_2\ (\mathit{vertices} \cdot \mathit{top})\ (\mathit{edges} \cdot \mathit{top})))$$

which allows conditions on the atoms and bonds to be combined. This rewrite gives the flexibility we require.

Here are all the rewrites.

$$
\begin{aligned}
&top \rightarrowtail (subgraphs\ 3) \cdot (setExists_1\ (\wedge_2\ (vertices \cdot top)\ (edges \cdot top)))\\
&top \rightarrowtail (domCard\ top) \cdot (> 0)\\
&top \rightarrowtail \wedge_2\ (connects \cdot (msetExists_2\ top\ top))\ (edge \cdot top)\\
&top \rightarrowtail vertex \cdot (\wedge_3\ (projElement \cdot top)\ (projAtomType \cdot top)\ (\wedge_2\ (projCharge \cdot top)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (projCharge \cdot top)))\\
&top \rightarrowtail (= 1)\\
&top \rightarrowtail (= 2)\\
&top \rightarrowtail (= 3)\\
&top \rightarrowtail (= 7)\\
&top \rightarrowtail (= Br)\\
&\qquad\quad \vdots\\
&top \rightarrowtail (= S)\\
&top \rightarrowtail (= 3)\\
&\qquad\quad \vdots\\
&top \rightarrowtail (= 195)\\
&top \rightarrowtail (\leq -0.117)\\
&top \rightarrowtail (\leq -0.087)\\
&top \rightarrowtail (\leq 0.013)\\
&top \rightarrowtail (\leq 0.142)\\
&top \rightarrowtail (\geq -0.117)\\
&top \rightarrowtail (\geq -0.087)\\
&top \rightarrowtail (\geq 0.013)\\
&top \rightarrowtail (\geq 0.142)\\
&(> 0) \rightarrowtail (> 1)\\
&(> 1) \rightarrowtail (> 2).
\end{aligned}
$$

These rewrites give a search space of predicates which contains the predicates appearing in the induced definition for the application given in Subsection 5.10.

One final point of interest with this example concerns the presence of the rewrites

$$
\begin{aligned}
&(> 0) \rightarrowtail (> 1)\\
&(> 1) \rightarrowtail (> 2).
\end{aligned}
$$

The first of these rewrites turns a predicate such as $(domCard\ p) \cdot (> 0)$ into the stronger predicate $(domCard\ p) \cdot (> 1)$. Now we could instead obtain such predicates by explicitly having the rewrites

$$
\begin{aligned}
&top \rightarrowtail (domCard\ top) \cdot (> 0)\\
&top \rightarrowtail (domCard\ top) \cdot (> 1)\\
&top \rightarrowtail (domCard\ top) \cdot (> 2).
\end{aligned}
$$

However, this approach does not capture the implication relationships which hold between the predicates $(domCard\ p) \cdot (> 0)$, $(domCard\ p) \cdot (> 1)$, and $(domCard\ p) \cdot (> 2)$. The big advantage of using '$(> 0) \rightarrowtail (> 1)$' and '$(> 1) \rightarrowtail (> 2)$' instead is that these implication relationships are captured. The implication relationship can be exploited to perform pruning of the search space, as explained in the next section. This completes the example.

We conclude this section with some remarks about the potential contribution of this paper to feature extraction, construction, and selection. Any machine learning problem involves at the beginning a knowledge representation task – that of modelling in a suitable formalism the individuals of interest in the application. As we have seen from Sections 3 and 5 (and the Appendix), basic terms provide a powerful knowledge representation language for modelling individuals. The next task may be to establish a suitable set of features for the application. The approach of this section provides a framework for this. Depending on the type of the individual, there is a corresponding set of transformations. For a particular application, there are also likely to be additional domain-specific transformations. We showed in this section how to systematically generate predicates from the set of transformations by specifying a set of rewrites and, in the next section, we apply this to a decision-tree learning system. However, this approach can be applied much more generally. For example, suppose one had a learning system that required individuals to be represented by a feature tuple of boolean values. Then such a feature-tuple representation could be provided by enumerating a large set of predicates via some rewrite system, ordering the predicates according to some heuristic that measured their promise, and then taking the top 200 predicates, say, as the components of the feature tuple. More generally, the transformations appropriate to the application provide a set of components from which features can be constructed.

7 A Learning System

Having discussed our approach to knowledge representation for inductive learning, we now present a decision-tree learning system based on these ideas. The main motivation for this section is to support our claim that the approach to knowledge representation adopted in this paper provides a flexible, convenient and effective framework for inductive learning problems in which the individuals have complex structure. For this purpose, we develop here a decision-tree learning system, since this is a standard approach to inductive learning and fits well with our approach to knowledge representation. In the next section, we apply the learning system to two substantial benchmark applications. The decision-tree learning system does have another novel feature, namely, its use of the accuracy heuristic, in addition to its exploitation of higher-order logic as a knowledge representation language. However, we concentrate here on knowledge representation issues, particularly the advantages of using a rewrite system to specify the hypothesis language.

7.1 Accuracy Heuristic

Decision-tree learning systems have a long history of development and successful application [BFOS84, Mit97]. These systems are based on the following intuitively appealing idea. To build a classifier from a set of examples, find a criterion that splits the examples into $k \geq 2$ subsets which are purer in the distribution of classes they contain than the original set; apply this process recursively on the child nodes until the leaf nodes of the tree are sufficiently pure; and then use the resulting decision tree as the induced classifier. Widely-used systems based on this idea include CART [BFOS84], ID3 [Qui86], and C4.5 [Qui93]. Our formulation focuses on *binary* decision trees so that $k = 2$ and the splitting criterion is a predicate.

An important issue for (binary) decision-tree algorithms is to decide under what situations a node should be split and, if so, what predicate should be used to split it. Given that the overall aim of the algorithm is to produce a tree with high predictive accuracy, it is natural to propose to use accuracy as the criterion for node splitting. (The accuracy of a set of examples is the fraction of the examples in the majority class of the set.) Unfortunately, given the standard way that decision-tree algorithms generate predicates for splitting, accuracy turns out to be a poor heuristic, as is explained in some detail in [BFOS84, Ch.4]. The essential point is that the accuracy heuristic is a function which is v-shaped, whereas the widely used heuristics (such as entropy) are strictly concave functions. This means that it is much more difficult to raise accuracy by splitting a node than it is to lower entropy by the same process. (Lower values of entropy correspond to purer nodes.) Thus, with the standard way of generating predicates, the use of the accuracy heuristic does not allow the tree to grow to a sufficient size and hence to provide optimal predictive accuracy. Commonly, decision-tree algorithms use a strictly concave heuristic which allows the tree to grow quite large and then employ post-pruning techniques to reduce the size of the tree without lowering accuracy on the training set [Mit97].

However, these arguments are based on the implicit assumption that only very simple predicates, involving a single attribute, should be used for splitting. A typical such predicate has the form $proj_i \cdot (= C)$. In effect, the combination of single-attribute predicates and the accuracy heuristic allows little chance to look ahead and thus potentially useful splits can be missed. This assumption of single-attribute predicates is crucial to the argument in the sense that, if one is prepared to use more complicated predicates to split, the accuracy heuristic has a better chance of finding a good split. Given that we are committed to using complex predicates because we admit a variety of types other than tuples, we take this opportunity to revisit the question of what the heuristic should be and develop a decision-tree learning algorithm based on accuracy. We also exploit a proposition for the accuracy heuristic (Proposition 7.1 below) that safely reduces the search space of predicates for splitting a node. On the other hand, the issue of the heuristic is orthogonal to the knowledge representation issues introduced in this paper, so accuracy could be replaced by any other suitable heuristic, if desired.

We now give the theoretical background for the accuracy heuristic. First, we give some definitions. 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 *majority class* of $\mathcal{E}$ to be the class to which the greatest number of examples in $\mathcal{E}$ belong. (If there are two or more classes with the same maximal number of examples, one is chosen arbitrarily as the majority class.)

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 $\mathcal{E}$.

The accuracy is the fraction of examples which are correctly classified on the basis that the majority class gives the classification. $A_{\mathcal{E}}$ lies in the range $[1/c, 1]$, where larger values correspond intuitively to purer sets of examples. If all the examples come from the same class, then $A_{\mathcal{E}} = 1$. If the examples are evenly distributed amongst the classes, then $A_{\mathcal{E}} = 1/c$. We also define the accuracy of a partition of a set of examples.

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

$$A_{\mathcal{P}} \equiv \sum_{j=1}^{m} \frac{N_j}{N} A_{\mathcal{E}_j}.$$

Thus $A_{\mathcal{P}}$ is the weighted average of the accuracies of the individual sets of examples in the partition. $A_{\mathcal{P}}$ lies in the range $[1/c, 1]$, where larger values correspond to partitions that classify more accurately.

The decision-tree learning algorithm makes binary splits at each node in the tree, so we now develop some material about binary partitions that will be needed. 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 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

$$B_{\mathcal{P}} \equiv (\max_{i} \{n_i + \max_{k \neq i} n_{1,k}\})/N.$$

We will show below that $B_{\mathcal{P}}$ is an upper bound for the accuracy of any partition that is a refinement of $\mathcal{P}$ and there exists a refinement of $\mathcal{P}$ that has accuracy $B_{\mathcal{P}}$. The intuitive idea behind the definition of $B_{\mathcal{P}}$ is that the refinement of $\mathcal{P}$ having the greatest accuracy can be obtained by moving all examples in one class, say i_0, from $\mathcal{E}_1$ across to $\mathcal{E}_2$ making i_0 the majority class in (the new) $\mathcal{E}_2$ and leaving $\text{argmax}_{k \neq i_0}\ n_{1,k}$ as the majority class in (the new) $\mathcal{E}_1$. Here is an example to illustrate the concept of refinement bound. In the following, we denote by $(n_1, \ldots, n_c)$ a set of examples with n_i examples from the ith class, for $i = 1, \ldots, c$.

Example. Let $\mathcal{E} = (6, 9, 3, 2)$ and suppose $\mathcal{P} = \{(2, 1, 0, 2), (4, 8, 3, 0)\}$. Then $A_{\mathcal{P}} = 10/20$ and $B_{\mathcal{P}} = 11/20$. If $\mathcal{Q} = \{(0, 9, 0, 0), (6, 0, 3, 2)\}$, then $A_{\mathcal{Q}} = B_{\mathcal{Q}} = 15/20$.

The following proposition collects the basic properties of refinement bounds.

Proposition 7.1. *Let $\mathcal{E}$ be a set of examples and $\mathcal{P}$ a binary partition of $\mathcal{E}$.*
(a) If $\mathcal{P}'$ is a refinement of $\mathcal{P}$, then $A_{\mathcal{P}'} \leq B_{\mathcal{P}}$. In particular, $A_{\mathcal{P}} \leq B_{\mathcal{P}}$.
(b) If $\mathcal{P}'$ is a refinement of $\mathcal{P}$, then $B_{\mathcal{P}'} \leq B_{\mathcal{P}}$.
(c) There is a refinement $\mathcal{R}$ of $\mathcal{P}$ such that $A_{\mathcal{R}} = B_{\mathcal{P}}$.
(d) $A_{\mathcal{P}} = B_{\mathcal{P}}$ iff, for all refinements $\mathcal{P}'$ of $\mathcal{P}$, $A_{\mathcal{P}'} \leq A_{\mathcal{P}}$.

Proof. (a) Let $\mathcal{P}$ be $\{\mathcal{E}_1, \mathcal{E}_2\}$ and $\mathcal{P}'$ be $\{\mathcal{E}_1', \mathcal{E}_2'\}$, where $\mathcal{E}_1' \subseteq \mathcal{E}_1$. Let $n_{1,i}$ be the number of examples in $\mathcal{E}_1$ in the ith class and M_2' the index of the majority class of $\mathcal{E}_2'$. Consider the partition $\mathcal{Q}$ obtained from $\mathcal{P}'$ by moving across any remaining examples from class M_2' in $\mathcal{E}_1'$ to $\mathcal{E}_2'$. Clearly $A_{\mathcal{P}'} \leq A_{\mathcal{Q}}$, since the examples so moved will be correctly classified after the move. But $A_{\mathcal{Q}} \leq B_{\mathcal{P}}$, since $A_{\mathcal{Q}} \leq (\{n_{M_2'} + \max_{k \neq M_2'}\ n_{1,k}\})/N$ and $B_{\mathcal{P}} = (\max_i\ \{n_i + \max_{k \neq i}\ n_{1,k}\})/N$. Hence the result.

(b) Let $n_{1,i}'$ be the number of examples in $\mathcal{E}_1'$ in the ith class, for $i = 1, \ldots, c$. Clearly $\max_{k \neq i}\ n_{1,k}' \leq \max_{k \neq i}\ n_{1,k}$, for $i = 1, \ldots, c$. Hence $B_{\mathcal{P}'} \leq B_{\mathcal{P}}$.

(c) Let i_0 be an index for which the maximum in the definition of $B_{\mathcal{P}}$ is achieved. Construct the partition $\mathcal{R} = \{\mathcal{E}_1', \mathcal{E}_2'\}$ from $\mathcal{P}$ by moving all examples in class i_0 from $\mathcal{E}_1$ across to $\mathcal{E}_2$. We claim that the majority class of $\mathcal{E}_2'$ is i_0. Suppose not. Then there is another class i_1, say, such that $\mathcal{E}_2'$ contains strictly more examples of class i_1 than class i_0. There are two cases to consider. Suppose that $\text{argmax}_{k \neq i_0}\ n_{1,k} \neq i_1$. Thus $n_{i_1} + \max_{k \neq i_1}\ n_{1,k} > n_{i_0} + \max_{k \neq i_0}\ n_{1,k}$ and we have a contradiction to the definition of i_0. Otherwise, suppose that $\text{argmax}_{k \neq i_0}\ n_{1,k} = i_1$. Thus $n_{i_1} > n_{i_0} + \max_{k \neq i_0}\ n_{1,k}$ and we have a contradiction again. Thus i_0 is the majority class in $\mathcal{E}_2'$ and it follows that $A_{\mathcal{R}} = B_{\mathcal{P}}$.

(d) Suppose that $A_{\mathcal{P}} = B_{\mathcal{P}}$. If $\mathcal{P}'$ is a refinement of $\mathcal{P}$, then $A_{\mathcal{P}'} \leq B_{\mathcal{P}}$, by part (a). Hence $A_{\mathcal{P}'} \leq A_{\mathcal{P}}$. Conversely, suppose $A_{\mathcal{P}'} \leq A_{\mathcal{P}}$, for all refinements $\mathcal{P}'$ of $\mathcal{P}$. By part (c), there is a refinement $\mathcal{R}$ of $\mathcal{P}$ such that $A_{\mathcal{R}} = B_{\mathcal{P}}$. Hence $A_{\mathcal{P}} \leq B_{\mathcal{P}} = A_{\mathcal{R}} \leq A_{\mathcal{P}}$ and the result follows. □

Part (a) of the proposition 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 proposition, if $B_{\mathcal{Q}} \leq A_{\mathcal{P}}$, then the partition $\mathcal{Q}$ and all its refinements can be safely pruned. Here is an example to illustrate how this works.

Example. Let the set of examples be $(6, 9, 3, 2)$. Suppose the best partition found so far is $\mathcal{P} = \{(5, 3, 0, 2), (1, 6, 3, 0)\}$, which has accuracy $11/20$. Suppose that later on in the search the partition $\mathcal{Q} = \{(2,4,0,1),(4,5,3,1)\}$ is being investigated. Note that $B_{\mathcal{Q}} = 11/20$. Since $B_{\mathcal{Q}} \leq A_{\mathcal{P}}$, the proposition shows that $\mathcal{Q}$ and its refinements can be pruned.

On the other hand, consider the partition $\mathcal{R} = \{(6,5,3,2),(0,4,0,0)\}$, for which $B_{\mathcal{R}} = 15/20$. Thus $\mathcal{R}$ has refinements, which could be found by the system, whose accuracies exceed that of $\mathcal{P}$. (The refinements actually investigated depend upon the hypothesis language, of course.) Thus $\mathcal{R}$ should not be pruned.

The algorithm for constructing predicates carefully structures the search space of predicates at a node so that the implication relation between predicates is (partially) known. More precisely, the algorithm knows that if it applies a rewrite to a predicate b to form a new predicate b', then b' implies b. The algorithm remembers the best predicate found so far and uses this information to prune according to the proposition. The effect of this pruning is to greatly reduce the part of the search space that has to be searched. Note carefully that the pruning is in the *search space of predicates* for splitting a particular node *not* in the decision tree itself.

7.2 The Algorithm

Now we are in a position to give an overview of the decision-tree algorithm. (See Figure 7.) 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 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 $p \cdot \neg$. The algorithm terminates if the chosen node n cannot be split to raise the accuracy. During construction of the decision tree, its branches are labelled by predicates. At the end of the algorithm, each leaf node is labelled by its majority class. In Figure 7, *Examples*(m) is the set of examples associated with node m and *Leaves*$(tree)$ is the set of leaf nodes of the tree *tree*.

The learner has four parameters that can be set. The one directly relevant to Figure 7 is the parameter *splits*, defined as follows.

> *splits*: This parameter is a non-negative integer. It fixes an upper bound on the number of nodes in the decision tree. If *splits* is 0, then there is no such limit on the number of splits.

Typically, *splits* is 0 or 1. In the latter case, there can be just one split. On multiple-instance problems like the Musk problem to be discussed later, *splits* is set to 1. Generally, it is difficult to know in advance how many splits are likely to give a good result, so choices of the value of *splits* other than 0 or 1 are uncommon.

The most interesting part of the algorithm is the way it searches the space of predicates to find a good predicate to split a node. This enumeration process was described earlier in Section 6 and the function *Predicate* given in Figure 8 is a variation of the earlier enumeration algorithm in Figure 6 that instead returns a single predicate. The function *Predicate* makes use of the other three parameters available for the learner. These parameters are as follows.

> *prune*: This parameter is a percentage. Only predicates that induce a partition with a refinement bound greater than or equal to *prune* are put on the open list for later expansion.
>
> *cutout*: This parameter is a non-negative integer. If the algorithm investigates successively *cutout* predicates without finding one which strictly improves the current best accuracy, then the algorithm terminates, returning the best predicate found so far. If *cutout* is 0, then no such limit is set.
>
> *balance*: This parameter is a non-negative integer. If the number of examples in either the left or right node of a split is less than *balance*, then the predicate inducing the split is not eligible to be returned.

function $Learn(\mathcal{E}, \rightarrowtail, \mathcal{S}, P, C, B, S)$ **returns** a decision tree;
inputs: $\mathcal{E}$, a set of examples;
$\rightarrowtail$, a regular, implication-consistent relation;
$\mathcal{S}$, a set of skeletons together with their monotone arguments;
P, *prune* parameter;
C, *cutout* parameter;
B, *balance* parameter;
S, *splits* parameter;

tree := single node (with examples $\mathcal{E}$);
finished := *false*;
while *not finished* **do**
$n := argmin_{m \in Leaves(tree)} A_{Examples(m)}$;
$\mathcal{F} := Examples(n)$;
$p := Predicate(\mathcal{F}, \rightarrowtail, \mathcal{S}, P, C, B)$;
$\mathcal{P}$:= partition of $\mathcal{F}$ induced by p;
if $A_{\mathcal{P}} > A_{\mathcal{F}}$
then
tree := extension of *tree* by two nodes under n corr. to p and $p \cdot \neg$;
$S := S - 1$;
finished := $(S = 0)$;
else
finished := *true*;
label each leaf node of *tree* by its majority class;
return *tree*;

Figure 7: Decision-tree algorithm

The parameter *prune* is important. It causes predicates whose induced partitions do not have a refinement bound larger than *prune* to be pruned and thus removes predicates that do not have the potential for achieving splits of high accuracy. Typically, *prune* is set high, even 100% is common.

The open list is decreasingly ordered by the refinement bounds of the partitions induced by the predicates. (Predicates that have the same refinement bound are decreasingly ordered by accuracy.) Thus the predicate with the highest such value is at the head of the list. The refinement bound plays a crucial part in directing the search towards promising predicates, that is, those which have the potential for being strengthened to produce splits with high accuracy. In this regard, in Figure 8, the function *Insert* takes a predicate and the open list as arguments and returns a new open list with the predicate inserted in the appropriate place according to the ordering imposed by refinement bound.

The algorithm in Figure 7 terminates when the least accurate leaf node cannot be split to improve accuracy. Another version terminates only when *no* leaf node can be split to improve accuracy, and thus produces trees no smaller than those produced by the algorithm in Figure 7. A related issue not discussed here is that of post-pruning the trees to improve generalisation ability. Such techniques can be added in the standard way to the algorithm. For the two larger applications discussed in Section 8, post-pruning turns out not to be necessary. (In the case of the Musk problem, this is because only one split is allowed. In the case of the Mutagenesis problem,

```
function Predicate(ℰ, ↣, 𝒮, P, C, B) returns a predicate;
input: ℰ, a set of examples;
       ↣, a regular, implication-consistent relation;
       𝒮, a set of skeletons together with their monotone arguments;
       P, prune parameter;
       C, cutout parameter;
       B, balance parameter;

openList := [top];
seenSet := {top};
predicate := top;
accuracy := A_ℰ;
count := C;
while openList ≠ [] do
    p := head(openList);
    openList := tail(openList);
    for each r such that r ∈ Positions(p, 𝒮) and r ↣ s, for some s, do
        q := Regularise(p[r/s]);
        if q ∉ seenSet
        then
            seenSet := seenSet ∪ {q};
            𝒫 := partition of ℰ induced by q;
            count := count − 1;
            if (count = 0) then return predicate;
            if |{e ∈ ℰ : (q e)}| ≥ B
            then
              if A_𝒫 > accuracy ∧ |{e ∈ ℰ : (q · ¬ e)}| ≥ B
              then
                predicate := q;
                accuracy := A_𝒫;
                count := C;
                if A_𝒫 > P then P := A_𝒫;
              if B_𝒫 ≥ P ∧ A_𝒫 < B_𝒫
              then
                openList := Insert(q, openList);
return predicate;
```

Figure 8: Algorithm for finding a predicate to split a node

we got good results without having to resort to post-pruning.)

As we propose using rather expressive hypothesis languages, we now discuss the important issue of the computational complexity of the decision-tree algorithm given in Figures 7 and 8. Basically, the algorithm is performing one operation, that of testing whether an individual satisfies a predicate or not, over and over again. Everything else in the algorithm is much less expensive than the total cost of all these tests. Thus a useful measure of the computational complexity of the algorithm is the number N of times it tests whether an individual satisfies a predicate during the construction of the decision tree.

Let

$|\mathcal{E}|$ be the number of examples,
d the number of splits in the decision tree, and
p the maximum number of predicates generated to find a split.

Then

$$N \leq d \times |\mathcal{E}| \times p.$$

Assuming that a bound is placed on d, the computational complexity of the algorithm is thus $O(|\mathcal{E}| \times p)$.

The value of p is, of course, bounded above by the size of the complete space of predicates generated by the rewrites. This size can be calculated given the set of rewrites. As an example, consider the following. Let $\mu_1, \ldots, \mu_k$ be types, where there are n_i constants of type μ_i, for $i = 1, \ldots, k$. Let the transformations be $(= C)$, for each constant C, the projections proj_i from the product $\mu_1 \times \cdots \times \mu_k$ onto the component μ_i, and the conjunction transformation $\wedge_k$. Let the set of rewrites be

$$\mathit{top} \rightarrowtail \wedge_k\ (\mathit{proj}_1 \cdot \mathit{top}) \cdots (\mathit{proj}_k \cdot \mathit{top}),$$

together with all rewrites of the form

$$\mathit{top} \rightarrowtail (= C),$$

for each constant C. Then there are $(n_1 + 1) \times \cdots \times (n_k + 1)$ predicates generated by the rewrites. (The extra one for each component comes from the predicate *top*, in addition to all the $(= C)$.)

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 the parameter *prune* to artificially raise 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 predictive power 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.

8 Two Larger Applications

In this section, we investigate the application of the learning system to two well-known benchmark applications, the Musk problem and the Mutagenesis problem. But first there is a short subsection outlining the key elements of ROC analysis that we use later to compare various classifiers.

8.1 ROC Analysis

In [PF01], a methodology for comparing learning algorithms is presented that has the benefit of separating the issues of classifier performance from specific class and cost distributions. The basis of this analysis is an ROC diagram, which comes from earlier work mainly in signal processing and medical diagnosis, the acronym ROC standing for 'Receiver Operating Characteristic'. The setting, following the presentation and notation of [PF01], is as follows.

We consider just two-class classification problems, and let $\mathbf{p}$ denote the positive class and $\mathbf{n}$ the negative class. To distinguish the actual from the predicted classes, we let $\mathbf{Y}$ and $\mathbf{N}$ be the classifications produced by the model. The prior probabilities of the classes are $p(\mathbf{p})$ and $p(\mathbf{n})$. Let the *true positive rate*, TP, be defined by

$$TP = p(\mathbf{Y}|\mathbf{p}) \approx \frac{\text{positives correctly classified}}{\text{total positives}},$$

and the *false positive rate*, FP, be defined by

$$FP = p(\mathbf{Y}|\mathbf{n}) \approx \frac{\text{negatives incorrectly classified}}{\text{total negatives}}.$$

An ROC diagram has FP on the x-axis and TP on the y-axis. For each classifier with a boolean output, there corresponds a point in the ROC diagram depending on the false positive rate and true positive rate of the classifier. (For classifiers with continuous output, one gets a boolean classifier for each choice of the threshold and a corresponding continuous curve on the ROC diagram – we are not concerned with such classifiers in this paper.) In Figure 9 below, three classifiers A, B and C are shown in the ROC diagram.

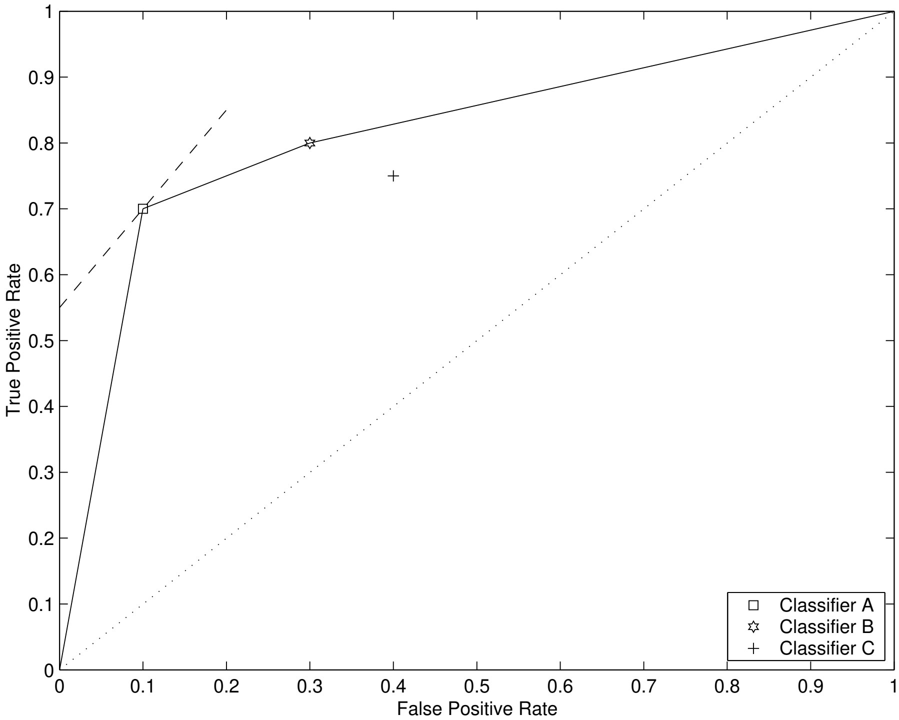

Figure 9: An ROC diagram

The next step is to introduce the class distribution and misclassification costs. Let $c(\mathbf{Y}, \mathbf{n})$ be the cost of a false positive error and $c(\mathbf{N}, \mathbf{p})$ be the cost of a false negative error. Then the expected cost of applying the classifier represented by a point (FP, TP) in the ROC diagram is

$$p(\mathbf{p}) \cdot (1 - TP) \cdot c(\mathbf{N}, \mathbf{p}) \ + \ p(\mathbf{n}) \cdot FP \cdot c(\mathbf{Y}, \mathbf{n}).$$

Thus two classifiers given by (FP_1, TP_1) and (FP_2, TP_2) have the same performance if

$$\frac{TP_2 - TP_1}{FP_2 - FP_1} = \frac{c(\mathbf{Y}, \mathbf{n}) \cdot p(\mathbf{n})}{c(\mathbf{N}, \mathbf{p}) \cdot p(\mathbf{p})}.$$

This equation defines the gradient of an *iso-performance line*, that is, all classifiers corresponding to points on this line have the same performance. Each choice of class and cost distributions defines a family of iso-performance lines, where those lines having a higher TP intercept are better because they have a lower cost. (This follows directly from the cost function.)

It is then shown in [PF01] how to identify the potentially optimal classifiers. The idea is to form the convex hull of the points in the ROC diagram corresponding to the various classifiers (including the point (0, 0) corresponding to the classifier which classifies all individuals as negative and the point (1, 1) corresponding to the classifier which classifies all individuals as positive). In Figure 9, the convex hull corresponds to all points on or below the solid piecewise-linear curve from (0, 0) to (1, 1). Points on the edge of this convex hull, that is, points on the piecewise-linear curve, correspond to potentially optimal classifiers. (Here, 'optimal' means 'optimal relative to the set of classifiers under consideration'.) For a particular choice of class and cost distributions, one chooses the optimal classifier to be the point on the edge at which a line with gradient determined by the given class and cost distributions touches the edge. (The case where there is a line segment on the edge parallel to the iso-performance line is discussed in [PF01].) In Figure 9, the dashed line segment has a gradient given by some chosen class and cost distributions and touches the edge at the point corresponding to classifier A. Thus classifier A is optimal (amongst the considered classifiers) for this particular choice. Classifier B is optimal for some other choices of class and cost distributions, while classifier C is never optimal as it lies strictly inside the convex hull.

8.2 Musk Problem

The first application is the Musk problem introduced in [DLLP97]. The main interest in this problem is that it is a multiple-instance problem with which conventional learners have difficulty. Indeed, in [DLLP97], the authors developed several special-purpose algorithms for solving the problem and showed that conventional learners performed badly. This problem is of interest here because 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 that 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 162 of these numbers represent the distance in angstroms from some origin in the conformation out along a radial line to the surface of the conformation and the other four numbers represent the position of a specific oxygen atom.

In the original musk dataset, all 166 attributes are continuous values. This data was discretised as follows. For each attribute, we calculated the mean μ and the standard deviation σ of the values occurring in the data. We then built up intervals centered on the mean, taking the width of each interval to be one standard deviation, and assigned integral labels to the intervals, so that interval 0 centred on the mean is $[\mu - \sigma/2, \mu + \sigma/2]$, interval 1 is $[\mu + \sigma/2, \mu + 3\sigma/2]$, interval -1 is $[\mu - 3\sigma/2, \mu - \sigma/2]$, and so on. We chose to use 13 intervals in total, labeled -6 through 6, as this covers most of the distribution and gives adequate resolution. The outermost intervals, -6 and 6, were extended below and above respectively to cover any outlying points.

The projections are named $proj_1, \ldots, proj_{166}$. Thus a predicate such as $proj_{98} \cdot (= 1)$ 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.

$$
\begin{aligned}
&-6, -5, \ldots, 5, 6 : \mathit{Distance} \\
&\mathit{Conformation} = \mathit{Distance} \times \cdots \times \mathit{Distance} \\
&\mathit{Molecule} = \{\mathit{Conformation}\}.
\end{aligned}
$$

Here the product type $\mathit{Distance} \times \cdots \times \mathit{Distance}$ contains 166 components. We want to learn the following function.

$$\mathit{musk} : \mathit{Molecule} \to \Omega.$$

The hypothesis language contains the following transformations.

$$
\begin{aligned}
&(= -6) : \mathit{Distance} \to \Omega \\
&\qquad\quad \vdots \\
&(= 6) : \mathit{Distance} \to \Omega \\
&(\neq -6) : \mathit{Distance} \to \Omega \\
&\qquad\quad \vdots \\
&(\neq 6) : \mathit{Distance} \to \Omega \\
&\mathit{proj}_1 : \mathit{Conformation} \to \mathit{Distance} \\
&\qquad\quad \vdots \\
&\mathit{proj}_{166} : \mathit{Conformation} \to \mathit{Distance} \\
&\mathit{setExists}_1 : (\mathit{Conformation} \to \Omega) \to \mathit{Molecule} \to \Omega \\
&\wedge_3 : (\mathit{Conformation} \to \Omega) \to (\mathit{Conformation} \to \Omega) \to (\mathit{Conformation} \to \Omega) \to \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Conformation} \to \Omega \\
&\qquad\quad \vdots \\
&\wedge_9 : (\mathit{Conformation} \to \Omega) \to \cdots \to (\mathit{Conformation} \to \Omega) \to \mathit{Conformation} \to \Omega.
\end{aligned}
$$

We performed a substantial number of experiments with various collections of rewrites, but only the most interesting of these are reported here. A major issue is the number of conjuncts in the predicate in the argument to $\mathit{setExists}_1$. We experimented with the number of conjuncts ranging from 3 to 9. The search spaces, even in the case of just 3 conjuncts, are very large. For example, for 3 conjuncts there are about 13 billion predicates.

For such large search spaces, one needs substantial computing power and for this reason we used a cluster machine, called bunyip, for the experiments. (This machine has achieved some prominence as another project based on it won the 2000 Gordon Bell Prize for price/performance for a supercomputer.) It consists of 96 dual Pentium III CPUs running at 550MHz, together with two server nodes, and the network is based on multiple 100Mbps Ethernet links and a novel arrangement of Ethernet switches. Detailed information about bunyip is available at `http://tux.anu.edu.au/Projects/Beowulf/`.

Parallelising the decision-tree algorithm in Figures 7 and 8 for a cluster computer turns out to be rather straightforward as one can partition the search space of predicates into a number of subspaces that can be handled by individual processors with minimal communication between the processors. We do not go into the details of the parallelisation here but do note that it was greatly facilitated by our rewrite system approach to predicate construction. We created a system of rewrites so that those for the top-level *top* partitioned the search space into nearly 14000 subproblems that could be farmed out to the individual processors in bunyip.

Here are the rewrites for the 3-conjunct case.

$$
\begin{array}{l}
top \rightarrowtail setExist_1\ (\wedge_3\ (proj_1 \cdot top)\ (proj_2 \cdot top)\ top) \\
top \rightarrowtail setExist_1\ (\wedge_3\ (proj_1 \cdot top)\ (proj_3 \cdot top)\ top) \\
\qquad\qquad \vdots \\
top \rightarrowtail setExist_1\ (\wedge_3\ (proj_{164} \cdot top)\ (proj_{166} \cdot top)\ top) \\
top \rightarrowtail setExist_1\ (\wedge_3\ (proj_{165} \cdot top)\ (proj_{166} \cdot top)\ top) \\
top \rightarrowtail proj_1 \cdot (= -6) \\
top \rightarrowtail proj_1 \cdot (= -5) \\
\qquad\qquad \vdots \\
top \rightarrowtail proj_{166} \cdot (\neq 5) \\
top \rightarrowtail proj_{166} \cdot (\neq 6) \\
top \rightarrowtail (= -6) \\
top \rightarrowtail (= -5) \\
\qquad\qquad \vdots \\
top \rightarrowtail (\neq 5) \\
top \rightarrowtail (\neq 6).
\end{array}
$$

The first group of ${}^{166}C_2 = 13695$ rewrites apply to the (top-level) redex *top* of type $Molecule \rightarrow \Omega$. Each of the top-level rewrites has three *top*s in them. The first two have type $Distance \rightarrow \Omega$ and have 26 possible rewrites that could be applied to them (the last group of rewrites). The third *top* has type $Conformation \rightarrow \Omega$ and has $166 \times 26 = 4316$ possible rewrites that could be applied (the second group of rewrites). The complete search space for the 3-conjunct case contains ${}^{166}C_3 \times 26^3 = 1.316 \times 10^{10}$ predicates.

The top-level rewrites can be used to partition the search space. For example, the first rewrite defines the subspace of the entire search space containing predicates in which there are projections onto the first and second components of a conformation, while the second rewrite defines the subset containing predicates in which there are projections onto the second and third components of a conformation. Essentially, each processor works with a rewrite set consisting of just one of the top-level rewrites together with all the remaining rewrites. The coordination of the entire search process is carried out by a master processor that hands out top-level predicates to other processors and records the discovery of new (globally) best predicates, broadcasting information about the accuracy of the current (globally) best predicate and increases to the prune parameter P as required.

Each of the 13695 subproblems has a large search space itself, averaging around 960821 predicates each. It is not feasible to exhaustively search these subspaces even with safe pruning, so we pruned more aggressively by adopting a prune parameter of 100% (which prunes all successors of predicates whose refinement bounds are less than 100%) and we used a cutout parameter (which terminates the search if no better predicate is found within the specified number of constructed predicates) of 15000 and 25000 in the reported experiments. One can think of the search process as being the totality of comparatively local searches taking place at 13695 sites in the full search space.

As an aside we note that the rewrites given above were chosen because breaking down the problem into around 14000 subproblems suited the parallelisation on bunyip well. But there are also other possibilities. For example, on a single-processor machine for which there is no advantage in a top-level partitioning into subproblems, one might use the following set.

$$
\begin{array}{l}
top \rightarrowtail setExists_1\ (\wedge_3\ top\ top\ top) \\
top \rightarrowtail proj_1 \cdot (= -6)
\end{array}
$$

$$top \rightarrowtail proj_1 \cdot (= -5)$$
$$\vdots$$
$$top \rightarrowtail proj_{166} \cdot (\neq 5)$$
$$top \rightarrowtail proj_{166} \cdot (\neq 6).$$

Alternatively, one might use the following rewrites which produces a much finer-grain partitioning of the search space.

$$top \rightarrowtail setExists_1 \ (\wedge_3 \ (proj_1 \cdot top) \ (proj_2 \cdot top) \ (proj_3 \cdot top))$$
$$top \rightarrowtail setExists_1 \ (\wedge_3 \ (proj_1 \cdot top) \ (proj_3 \cdot top) \ (proj_4 \cdot top))$$
$$\vdots$$
$$top \rightarrowtail setExists_1 \ (\wedge_3 \ (proj_{163} \cdot top) \ (proj_{165} \cdot top) \ (proj_{166} \cdot top))$$
$$top \rightarrowtail setExists_1 \ (\wedge_3 \ (proj_{164} \cdot top) \ (proj_{165} \cdot top) \ (proj_{166} \cdot top))$$
$$top \rightarrowtail (= -6)$$
$$top \rightarrowtail (= -5)$$
$$\vdots$$
$$top \rightarrowtail (\neq 5)$$
$$top \rightarrowtail (\neq 6).$$

With the choice of rewrites for the 3-conjunct case as stated above, experiments were carried out on the musk2 data set in [DLLP97] that contains 102 examples, of which 39 are musk and 63 are not. The 102 molecules have a total of 6598 conformations. (Earlier we had carried out experiments on the less demanding musk1 data set with good results.) For the first experiment reported here, the settings for the learner were $prune = 100\%$, $cutout = 15000$, $balance = max(2, 2\% \text{ of examples})$, and $splits = 1$. We ran three 10-fold cross-validations with these settings where each of the three resulted from a random partition of the training set into 10 subsets. The results from each of the three cross-validations were then averaged. Using 160 of bunyip's processors, each fold of each cross-validation took about 60 minutes, so the whole experiment took about 30 hours. The results (averaged over the three partitions) were as follows: accuracy = 82.2%, $FP = 0.16$ and $TP = 0.76$. The point on the ROC diagram corresponding to (0.16, 0.76) is given in Figure 10 as the '3-conjunct, 15000 cutout' classifier.

With the same settings, the learner found the following definition for the function *musk* on the whole data set of 102 examples.

$$\begin{aligned} &musk\ m = \\ &\quad if\ setExists_1\ (\wedge_3\ (proj_{29} \cdot (\neq -4))\ (proj_{119} \cdot (= 1))\ (proj_{132} \cdot (= -2)))\ m \\ &\quad then\ \top \\ &\quad else\ \bot. \end{aligned}$$

There was a lot of commonality amongst the definitions produced by the cross-validations and the definition just given. In fact, 25 of the 30 folds produced the above theory.

The next experiment was the same as the previous one except that the cutout parameter was raised to 25000. This meant that more searching was done in each of the 13695 subspaces. The results in this case were as follows: accuracy = 83.5%, $FP = 0.14$ and $TP = 0.78$. The point on the ROC diagram corresponding to (0.14, 0.78) is given in Figure 10 as the '3-conjunct, 25000 cutout' classifier. The theory produced on the entire set of examples was the same as for the 15000 cutout case and this time that same theory was produced in 27 of the 30 folds.

We also experimented with larger numbers of conjuncts. For four conjuncts, the top-level

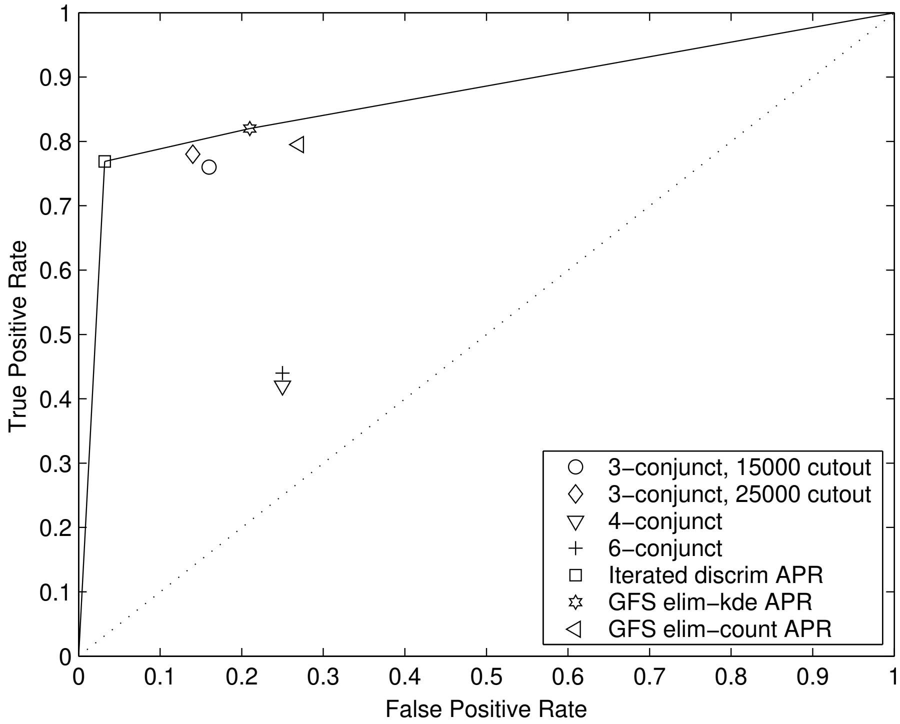

Figure 10: ROC diagram for musk experiments

rewrites were as follows.

$$top \rightarrowtail setExists_1\ (\wedge_4\ (proj_1 \cdot top)\ (proj_2 \cdot top)\ top\ top)$$
$$top \rightarrowtail setExists_1\ (\wedge_4\ (proj_1 \cdot top)\ (proj_3 \cdot top)\ top\ top)$$
$$\vdots$$
$$top \rightarrowtail setExists_1\ (\wedge_4\ (proj_{164} \cdot top)\ (proj_{166} \cdot top)\ top\ top)$$
$$top \rightarrowtail setExists_1\ (\wedge_4\ (proj_{165} \cdot top)\ (proj_{166} \cdot top)\ top\ top).$$

In this case, there are ${}^{166}C_4 \times 26^4 = 1.394 \times 10^{13}$ predicates in the search space. The settings for the learner were *prune* = 100%, *cutout* = 15000, *balance* = *max*(2, 2% of examples), and *splits* = 1. The results were as follows: accuracy = 60.6%, *FP* = 0.25 and *TP* = 0.42. The point on the ROC diagram corresponding to (0.25, 0.42) is given in Figure 10 as the '4-conjunct' classifier. With these settings, the definition for the function *musk* on the whole data set of 102 examples was as follows.

$$\begin{aligned}
&musk\ m = \\
&\quad if\ setExists_1\ (\wedge_4\ (proj_{29} \cdot (\neq -4))\ (proj_{94} \cdot (\neq -4))\ (proj_{119} \cdot (= 1))\ (proj_{132} \cdot (= -2)))\ m \\
&\quad then\ \top \\
&\quad else\ \bot.
\end{aligned}$$

This definition includes the three conjuncts from the 3-conjunct definition. The accuracy of this definition *on the whole data set* is 85.29%. Unfortunately, the accuracy from the cross-validations is very much lower because many of the folds find much less accurate definitions. The reason is that the search spaces for each of the top-level conjuncts is very much larger (by a factor of over 1000) than in the 3-conjunct case and so the learner has a much more difficult task finding an accurate definition. However, with some extra guidance built into the search process, there does appear to be a chance of raising the average accuracy from the cross-validations to around 85% with a 4-conjunct definition.

With 6 conjuncts, the results were: accuracy = 59.8%, $FP = 0.25$ and $TP = 0.44$. The point on the ROC diagram corresponding to (0.25, 0.44) is given in Figure 10 as the '6-conjunct' classifier. The results for 7, 8 and 9 conjuncts were similarly poor. Of course, the search spaces are also extraordinarily large in these cases.

In Figure 10, we have also given for comparison the best three results presented in [DLLP97]. The best of these is the 'Iterated discrim APR' classifier which has accuracy = 89.2%, $FP = 0.03$ and $TP = 0.77$. This classifier lies on the edge of the convex hull, as does the second best classifier in [DLLP97], and thus these classifiers are optimal for some class and cost distributions. Our best classifier, the '3-conjunct, 25000 cutout' one, is just off the edge of the convex hull and is therefore not optimal under any conditions. However, it performed well in comparison to the learners in [DLLP97], which have been optimised for this problem. Also, compared with other general-purpose learners, ours is highly competitive. For example, the learner in [BD97, BD98] is reported as having an accuracy of 79.4%. (The results for FP and TP are not given.)

We conclude that our higher-order approach to knowledge representation has led to the design and implementation of a general-purpose decision-tree learning system which is more effective than many other learners that have been applied to the musk problem and is only slightly less effective than the best learner optimised for this particular problem.

8.3 Mutagenesis Problem

The next example involves learning a theory to predict whether a chemical molecule is mutagenic or not. We refer the reader to [KMSS96], [SMKS94], and [Yor] for background material on this important and interesting problem.

As before, we use an (undirected) graph to model a molecule. The type *Element* is the type of the (relevant) chemical elements.

$$Br, C, Cl, F, H, I, N, O, S : Element.$$

We also make the following type synonyms.

$$\begin{aligned}
&I_1 = \Omega \\
&I_a = \Omega \\
&\epsilon_{LUMO} = Float \\
&AtomType = Nat \\
&Charge = Float \\
&Atom = Element \times AtomType \times Charge \\
&Bond = Nat \\
&Structure = Graph\ Atom\ Bond \\
&Molecule = I_1 \times I_a \times \epsilon_{LUMO} \times Structure.
\end{aligned}$$

A *Structure* is the same as what was called a *Molecule* in Subsection 5.10. Here a molecule has three additional attributes I_1, I_a and ϵ_{LUMO}, which are explained in [KMSS96]. For the molecule d_1 given earlier, I_1 is $\top$, I_a is $\bot$ and ϵ_{LUMO} is -1.246.

The examples are the set of 188 regression-friendly molecules, available at [Yor]. For instance, the molecule d_1 in this set is mutagenic. We want to learn the definition of the function *mutagenic* having the following signature.

$$mutagenic : Molecule \to \Omega.$$

The hypothesis language contains the following transformations.

$$
\begin{array}{l}
(=\top) : I_1 \to \Omega \\
(=\bot) : I_1 \to \Omega \\
(=\top) : I_a \to \Omega \\
(=\bot) : I_a \to \Omega \\
(\leq -3.718) : \epsilon_{LUMO} \to \Omega \\
(\leq -3.368) : \epsilon_{LUMO} \to \Omega \\
(\leq -3.168) : \epsilon_{LUMO} \to \Omega \\
(\leq -3.018) : \epsilon_{LUMO} \to \Omega \\
(\leq -2.668) : \epsilon_{LUMO} \to \Omega \\
(\leq -2.418) : \epsilon_{LUMO} \to \Omega \\
(\leq -0.718) : \epsilon_{LUMO} \to \Omega \\
(\geq -3.418) : \epsilon_{LUMO} \to \Omega \\
(\geq -3.218) : \epsilon_{LUMO} \to \Omega \\
(\geq -3.068) : \epsilon_{LUMO} \to \Omega \\
(\geq -2.918) : \epsilon_{LUMO} \to \Omega \\
(\geq -2.618) : \epsilon_{LUMO} \to \Omega \\
(\geq -2.368) : \epsilon_{LUMO} \to \Omega \\
(\geq -0.668) : \epsilon_{LUMO} \to \Omega \\
(= Br) : Element \to \Omega \\
\qquad\qquad \vdots \\
(= S) : Element \to \Omega \\
(= 1) : AtomType \to \Omega \\
(= 3) : AtomType \to \Omega \\
\qquad\qquad \vdots \\
(= 232) : AtomType \to \Omega \\
(\geq -0.781) : Charge \to \Omega \\
(\geq -0.424) : Charge \to \Omega \\
(\geq -0.067) : Charge \to \Omega \\
(\geq 0.290) : Charge \to \Omega \\
(\geq 0.647) : Charge \to \Omega \\
(\geq -0.424) : Charge \to \Omega \\
(\geq -0.067) : Charge \to \Omega \\
(\geq 0.290) : Charge \to \Omega \\
(\geq 0.647) : Charge \to \Omega \\
(\geq 1.004) : Charge \to \Omega \\
(= 1) : Bond \to \Omega \\
(= 2) : Bond \to \Omega \\
(= 3) : Bond \to \Omega \\
(= 4) : Bond \to \Omega \\
(= 5) : Bond \to \Omega \\
(= 7) : Bond \to \Omega
\end{array}
$$

$$
\begin{aligned}
&(>0) : \mathit{Nat} \to \Omega \\
&(>1) : \mathit{Nat} \to \Omega \\
&(>2) : \mathit{Nat} \to \Omega \\
&(>3) : \mathit{Nat} \to \Omega \\
&(>4) : \mathit{Nat} \to \Omega \\
&\mathit{projI}_1 : \mathit{Molecule} \to I_1 \\
&\mathit{projI}_a : \mathit{Molecule} \to I_a \\
&\mathit{proj}\epsilon_{LUMO} : \mathit{Molecule} \to \epsilon_{LUMO} \\
&\mathit{projStructure} : \mathit{Molecule} \to \mathit{Structure} \\
&\mathit{projElement} : \mathit{Atom} \to \mathit{Element} \\
&\mathit{projAtomType} : \mathit{Atom} \to \mathit{AtomType} \\
&\mathit{projCharge} : \mathit{Atom} \to \mathit{Charge} \\
&\mathit{vertices} : \mathit{Structure} \to \{\mathit{Vertex\ Atom\ Bond}\} \\
&\mathit{edges} : \mathit{Structure} \to \{\mathit{Edge\ Atom\ Bond}\} \\
&\mathit{vertex} : \mathit{Vertex\ Atom\ Bond} \to \mathit{Atom} \\
&\mathit{edge} : \mathit{Edge\ Atom\ Bond} \to \mathit{Bond} \\
&\mathit{connects} : \mathit{Edge\ Atom\ Bond} \to (\mathit{Vertex\ Atom\ Bond} \to \mathit{Nat}) \\
&\mathit{domCard} : (\mathit{Vertex\ Atom\ Bond} \to \Omega) \to \{\mathit{Vertex\ Atom\ Bond}\} \to \mathit{Nat} \\
&\mathit{domCard} : (\mathit{Edge\ Atom\ Bond} \to \Omega) \to \{\mathit{Edge\ Atom\ Bond}\} \to \mathit{Nat} \\
&\mathit{msetExists}_2 : (\mathit{Vertex\ Atom\ Bond} \to \Omega) \to (\mathit{Vertex\ Atom\ Bond} \to \Omega) \to \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Vertex\ Atom\ Bond} \to \mathit{Nat}) \to \Omega \\
&\wedge_2 : (\mathit{Charge} \to \Omega) \to (\mathit{Charge} \to \Omega) \to \mathit{Charge} \to \Omega \\
&\wedge_2 : (\mathit{Atom} \to \Omega) \to (\mathit{Atom} \to \Omega) \to \mathit{Atom} \to \Omega \\
&\wedge_3 : (\mathit{Atom} \to \Omega) \to (\mathit{Atom} \to \Omega) \to (\mathit{Atom} \to \Omega) \to \mathit{Atom} \to \Omega \\
&\wedge_2 : (\mathit{Molecule} \to \Omega) \to (\mathit{Molecule} \to \Omega) \to \mathit{Molecule} \to \Omega \\
&\wedge_3 : (\mathit{Molecule} \to \Omega) \to (\mathit{Molecule} \to \Omega) \to (\mathit{Molecule} \to \Omega) \to \mathit{Molecule} \to \Omega \\
&\wedge_4 : (\mathit{Molecule} \to \Omega) \to (\mathit{Molecule} \to \Omega) \to (\mathit{Molecule} \to \Omega) \to (\mathit{Molecule} \to \Omega) \to \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Molecule} \to \Omega.
\end{aligned}
$$

We performed numerous experiments with various sets of rewrites, five of which are reported here. All the experiments were run on a (single processor) Pentium III CPU running at 700MHz. The first of these experiments involved leaving out the structural information and using just the attributes I_1, I_a and ϵ_{LUMO}. For this, each molecule is simply represented by a triple of values for these three attributes.

$$\mathit{Molecule} = I_1 \times I_a \times \epsilon_{LUMO}.$$

The rewrites were as follows.

$$
\begin{aligned}
&\mathit{top} \rightarrowtail \wedge_3\ (\mathit{projI}_1 \cdot \mathit{top})\ (\mathit{projI}_a \cdot \mathit{top})\ (\mathit{proj}\epsilon_{LUMO} \cdot \mathit{top}) \\
&\mathit{top} \rightarrowtail (= \top) \\
&\mathit{top} \rightarrowtail (= \bot) \\
&\mathit{top} \rightarrowtail (\leq -3.718) \\
&\qquad\qquad \vdots \\
&\mathit{top} \rightarrowtail (\geq -0.668).
\end{aligned}
$$

In this context, the learner searched for a definition for *mutagenic* with the settings $\mathit{prune} = 100\%$, $\mathit{cutout} = 0$, $\mathit{balance} = \mathit{max}(2, 2\%$ of examples$)$, and $\mathit{splits} = 0$. On a 10-fold cross-validation (taking about 10 seconds), the average accuracy was 87.2% (very similar to the results given in [SMKS94, KMSS96, SKM99, BD97, BD98]), and $FP = 0.07$ and $TP = 0.84$. This classifier is the one called 'Indicators' in Figure 11. The learner found the following definition on the full data set. (In fact, this theory was also produced for each of the 10 training sets in the cross-validation.)

$$\begin{array}{l} \mathit{mutagenic}\ m = \\ \quad \mathit{if}\ \wedge_2\ (\mathit{projI}_1 \cdot (= \bot))\ (\mathit{proj}\epsilon_{LUMO} \cdot (\geq -2.368))\ m \\ \quad \mathit{then}\ \bot \\ \quad \mathit{else}\ \top. \end{array}$$

"A molecule is mutagenic iff I_1 is true or $\epsilon_{LUMO} < -2.368$".

Note that the definition that is produced directly by the learner actually has the condition $\wedge_3\ (\mathit{projI}_1 \cdot (= \bot))\ (\mathit{projI}_a \cdot \mathit{top})\ (\mathit{proj}\epsilon_{LUMO} \cdot (\geq -2.368))$ instead of the equivalent $\wedge_2$ condition above. The simpler form can be obtained by some straightforward post-processing and it makes the definitions easier to read, so we do this without comment in the following experiments.

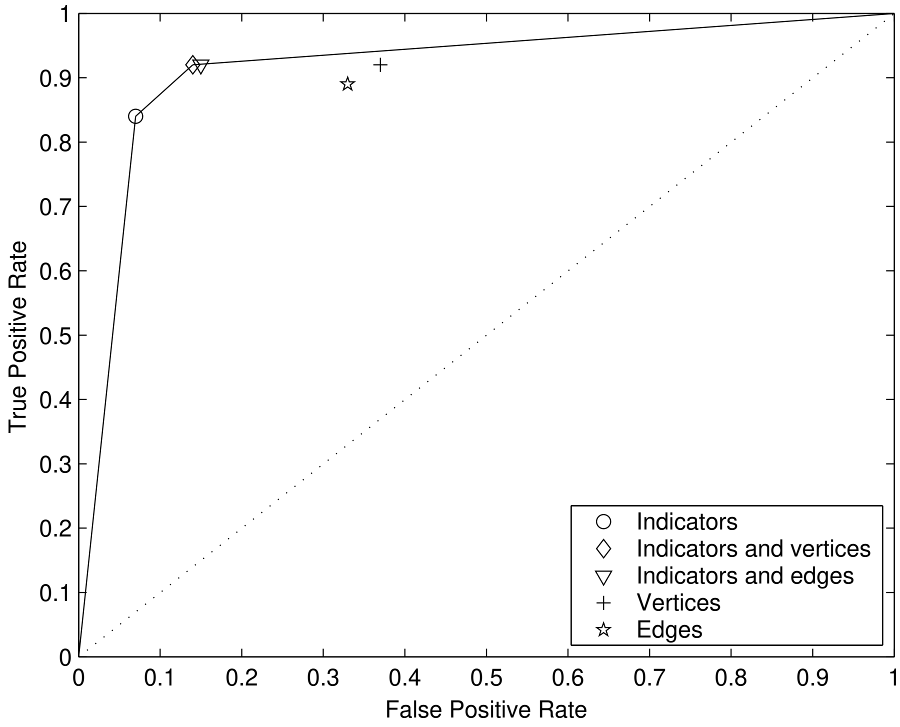

Figure 11: ROC diagram for mutagenesis experiments

For the second experiment, we included information about the atoms in the molecules. Thus a molecule was represented as a 4-tuple.

$$\mathit{Molecule} = I_1 \times I_a \times \epsilon_{LUMO} \times \mathit{Structure}.$$

The rewrites were as follows.

$$\begin{array}{l} \mathit{top} \rightarrowtail \wedge_4\ (\mathit{projI}_1 \cdot \mathit{top})\ (\mathit{projI}_a \cdot \mathit{top})\ (\mathit{proj}\epsilon_{LUMO} \cdot \mathit{top})\ (\mathit{projStructure} \cdot \mathit{top}) \\ \mathit{top} \rightarrowtail \mathit{vertices} \cdot (\mathit{domCard}\ (\mathit{vertex} \cdot \mathit{top})) \cdot (> 0) \\ \mathit{top} \rightarrowtail \mathit{projElement} \cdot \mathit{top} \end{array}$$

$$
\begin{aligned}
&top \rightarrowtail projAtomType \cdot top \\
&top \rightarrowtail projCharge \cdot top \\
&top \rightarrowtail \wedge_2\ (projAtomType \cdot top)\ (projCharge \cdot top) \\
&top \rightarrowtail (= \top) \\
&top \rightarrowtail (= \bot) \\
&top \rightarrowtail (\leq -0.718) \\
&(\leq -0.718) \rightarrowtail (\leq -2.418) \\
&\qquad\qquad \vdots \\
&(\leq -3.368) \rightarrowtail (\leq -3.718) \\
&top \rightarrowtail (\geq -3.418) \\
&(\geq -3.418) \rightarrowtail (\geq -3.218) \\
&\qquad\qquad \vdots \\
&(\geq -2.368) \rightarrowtail (\geq -0.668) \\
&top \rightarrowtail (= Br) \\
&\qquad\qquad \vdots \\
&top \rightarrowtail (= S) \\
&top \rightarrowtail (= 1) \\
&top \rightarrowtail (= 3) \\
&\qquad\qquad \vdots \\
&top \rightarrowtail (= 232) \\
&top \rightarrowtail (\geq -0.781) \\
&(\geq -0.781) \rightarrowtail (\geq -0.424) \\
&\qquad\qquad \vdots \\
&(\geq 0.647) \rightarrowtail (\geq 1.004) \\
&top \rightarrowtail (= 1) \\
&\qquad\qquad \vdots \\
&top \rightarrowtail (= 7) \\
&(> 0) \rightarrowtail (> 1) \\
&(> 1) \rightarrowtail (> 2) \\
&(> 2) \rightarrowtail (> 3) \\
&(> 3) \rightarrowtail (> 4).
\end{aligned}
$$

The learner searched for a definition for *mutagenic* with the settings $prune = 100\%$, $cutout = 0$, $balance = max(2, 2\%$ of examples$)$, and $splits = 0$. On a 10-fold cross-validation (taking about 5 hours), the average accuracy was 89.39%, and $FP = 0.14$ and $TP = 0.92$. This classifier is the one called 'Indicators and vertices' in Figure 11. The learner found the following definition on the full data set.

$$
\begin{array}{l}
\mathit{mutagenic}\ m = \\
\quad \mathit{if}\ \wedge_2\ (\mathit{projI}_1 \cdot (= \bot))\ (\mathit{proj}\epsilon_{LUMO} \cdot (\geq -2.368))\ m \\
\quad \mathit{then\ if}\ \mathit{projStructure} \cdot \mathit{vertices} \cdot (\mathit{domCard}\ (\mathit{vertex} \cdot \mathit{projAtomType} \cdot (= 38))) \cdot (> 1)\ m \\
\qquad \mathit{then\ if}\ \mathit{projStructure} \cdot \mathit{vertices} \cdot (\mathit{domCard}\ (\mathit{vertex} \cdot (\wedge_2 \\
\qquad\qquad\qquad (\mathit{projAtomType} \cdot (= 22))\ (\mathit{projCharge} \cdot (\geq -0 \cdot 067))))) \cdot (> 0)\ m \\
\qquad\quad \mathit{then}\ \bot \\
\qquad\quad \mathit{else}\ \top \\
\qquad \mathit{else}\ \bot \\
\quad \mathit{else}\ \top.
\end{array}
$$

"A molecule is mutagenic iff either

(i) I_1 is true, or

(ii) $\epsilon_{LUMO} < -2.368$, or

(iii) I_1 is false, $\epsilon_{LUMO} \geq -2.368$, it has at least two atoms of type 38, and it does not have an atom of type 22 and charge ≥ -0.067."

In the third experiment, we used information about the bonds instead of the atoms, as was used in the second experiment. The rewrites were as follows.

$$
\begin{array}{l}
\mathit{top} \rightarrowtail \wedge_4\ (\mathit{projI}_1 \cdot \mathit{top})\ (\mathit{projI}_a \cdot \mathit{top})\ (\mathit{proj}\epsilon_{LUMO} \cdot \mathit{top})\ (\mathit{projStructure} \cdot \mathit{top}) \\
\mathit{top} \rightarrowtail \mathit{edges} \cdot (\mathit{domCard}\ \mathit{top}) \cdot (> 0) \\
\mathit{top} \rightarrowtail \mathit{edge} \cdot \mathit{top} \\
\mathit{top} \rightarrowtail \mathit{connects} \cdot (\mathit{msetExists}_2\ \mathit{top}\ \mathit{top}) \\
\mathit{top} \rightarrowtail \mathit{vertex} \cdot \mathit{projAtomType} \cdot \mathit{top} \\
\mathit{top} \rightarrowtail (= \top) \\
\mathit{top} \rightarrowtail (= \bot) \\
\mathit{top} \rightarrowtail (\leq -3.718) \\
\qquad\quad \vdots \\
\mathit{top} \rightarrowtail (\leq -2.418) \\
\mathit{top} \rightarrowtail (\geq -3.418) \\
\qquad\quad \vdots \\
\mathit{top} \rightarrowtail (\geq -0.668) \\
\mathit{top} \rightarrowtail (= 1) \\
\mathit{top} \rightarrowtail (= 3) \\
\qquad\quad \vdots \\
\mathit{top} \rightarrowtail (= 232) \\
\mathit{top} \rightarrowtail (= 1) \\
\qquad\quad \vdots \\
\mathit{top} \rightarrowtail (= 7) \\
(> 0) \rightarrowtail (> 2) \\
(> 2) \rightarrowtail (> 4).
\end{array}
$$

The learner searched for a definition for *mutagenic* with the settings $prune = 100\%$, $cutout = 0$, $balance = max(2, 2\%$ of examples$)$, and $splits = 0$. On a 10-fold cross-validation (taking about 4 hours), the average accuracy was 88.86%, and $FP = 0.15$ and $TP = 0.92$. This classifier is the one called 'Indicators and edges' in Figure 11. The learner found the following definition on the full data set.

$$
\begin{array}{l}
\mathit{mutagenic}\ m = \\
\quad \mathit{if}\ \wedge_2\ (\mathit{projI}_1 \cdot (= \bot))\ (\mathit{proj}\epsilon_{LUMO} \cdot (\geq -2.368))\ m \\
\quad \mathit{then\ if}\ \mathit{projStructure} \cdot \mathit{edges} \cdot (\mathit{domCard}\ (\mathit{connects} \cdot (\mathit{msetExists}_2 \\
\qquad\qquad\qquad (\mathit{vertex} \cdot \mathit{projAtomType} \cdot (= 38))\ \mathit{top}))) \cdot (> 4)\ m \\
\qquad \mathit{then\ if}\ \mathit{projStructure} \cdot \mathit{edges} \cdot (\mathit{domCard}\ (\mathit{connects} \cdot (\mathit{msetExists}_2 \\
\qquad\qquad\qquad (\mathit{vertex} \cdot \mathit{projAtomType} \cdot (= 1))\ \mathit{top}))) \cdot (> 0)\ m \\
\qquad\quad \mathit{then}\ \bot \\
\qquad\quad \mathit{else\ if}\ \mathit{projStructure} \cdot \mathit{edges} \cdot (\mathit{domCard}\ (\mathit{connects} \cdot (\mathit{msetExists}_2 \\
\qquad\qquad\qquad (\mathit{vertex} \cdot \mathit{projAtomType} \cdot (= 50))\ \mathit{top}))) \cdot (> 0)\ m \\
\qquad\qquad \mathit{then}\ \bot \\
\qquad\qquad \mathit{else}\ \top \\
\qquad \mathit{else}\ \bot \\
\quad \mathit{else}\ \top.
\end{array}
$$

"A molecule is mutagenic iff either

(i) I_1 is true, or

(ii) $\epsilon_{LUMO} < -2.368$, or

(iii) I_1 is false, $\epsilon_{LUMO} \geq -2.368$, it has at least five atoms of type 38, it does not have an atom of type 1, and it does not have an atom of type 50."

As can be seen from the definition, the attempt to use bond information in this experiment failed. All the conditions are actually conditions on atoms rather than bonds.

As a fourth experiment, we used only the atom information, omitting I_1, I_a and ϵ_{LUMO}. In other words, we reverted to the earlier definition

$$\mathit{Molecule} = \mathit{Graph\ Atom\ Bond}.$$

The problem was now rather more difficult as I_1 and ϵ_{LUMO} are good indicators of mutagenicity, but were no longer available.

The consequent changes to the hypothesis language above were that projI_1, projI_a, $\mathit{proj}\epsilon_{LUMO}$, and *projStructure* were dropped, and *vertices* has domain of type *Molecule*. For the rewrites, those related to I_1, I_a, and ϵ_{LUMO} were dropped. We also dropped *edges* from the hypothesis language and the rewrites. The rewrites were as follows.

$$
\begin{array}{l}
\mathit{top} \rightarrowtail \mathit{vertices} \cdot (\mathit{domCard}\ (\mathit{vertex} \cdot \mathit{top})) \cdot (> 0) \\
\mathit{top} \rightarrowtail \mathit{projAtomType} \cdot \mathit{top} \\
\mathit{top} \rightarrowtail \mathit{projCharge} \cdot \mathit{top} \\
\mathit{top} \rightarrowtail \wedge_2\ (\mathit{projAtomType} \cdot \mathit{top})\ (\mathit{projCharge} \cdot \mathit{top}) \\
\mathit{top} \rightarrowtail (= 1) \\
\mathit{top} \rightarrowtail (= 3) \\
\qquad\quad \vdots \\
\mathit{top} \rightarrowtail (= 232)
\end{array}
$$

$$
\begin{aligned}
& top \rightarrowtail (\geq -0.781) \\
& (\geq -0.781) \rightarrowtail (\geq -0.424) \\
& \qquad\qquad \vdots \\
& (\geq 0.647) \rightarrowtail (\geq 1.004) \\
& (> 0) \rightarrowtail (> 1).
\end{aligned}
$$

The settings for the learner were $prune = 100\%$, $cutout = 0$, $balance = max(2, 2\%$ of examples), and $splits = 0$. On a 10-fold cross-validation (taking about 20 minutes), the average accuracy was 80.26%, and $FP = 0.37$ and $TP = 0.92$. This classifier is the one called 'Vertices' in Figure 11. The learner found the following definition on the full data set.

$$
\begin{aligned}
& mutagenic\, m = \\
& \quad if\ vertices \cdot (domCard\ (vertex \cdot (\wedge_2 \\
& \qquad\qquad\qquad (projAtomType \cdot (= 1))\ (projCharge \cdot (\geq 0 \cdot 290))))) \cdot (> 0)\ m \\
& \quad then\ if\ vertices \cdot (domCard\ (vertex \cdot (\wedge_2 \\
& \qquad\qquad\qquad (projAtomType \cdot (= 34))\ (projCharge \cdot (\geq -0 \cdot 424))))) \cdot (> 1)\ m \\
& \qquad then\ \top \\
& \qquad else\ \bot \\
& \quad else\ if\ vertices \cdot (domCard\ (vertex \cdot projAtomType \cdot (= 50))) \cdot (> 0)\ m \\
& \qquad then\ \bot \\
& \qquad else\ if\ vertices \cdot (domCard\ (vertex \cdot projAtomType \cdot (= 34))) \cdot (> 1)\ m \\
& \qquad\quad then\ \bot \\
& \qquad\quad else\ if\ vertices \cdot (domCard\ (vertex \cdot projAtomType \cdot (= 35))) \cdot (> 0)\ m \\
& \qquad\qquad then\ \bot \\
& \qquad\qquad else\ \top.
\end{aligned}
$$

"A molecule is mutagenic iff either

(i) it has at least one atom of type 1 and charge ≥ 0.290 and at least two atoms of type 34 and charge ≥ -0.424, or

(ii) it does not have an atom of type 1 and charge ≥ 0.290, it does not have an atom of type 50, it has at most one atom of type 34, and it does not have an atom of type 35."

The final mutagenesis experiment is similar to the previous one except we looked for predicates involving bonds. The rewrites were as follows.

$$
\begin{aligned}
& top \rightarrowtail edges \cdot (domCard\ top) \cdot (> 0) \\
& top \rightarrowtail edge \cdot top \\
& top \rightarrowtail connects \cdot (msetExists_2\ top\ top) \\
& top \rightarrowtail vertex \cdot projAtomType \cdot top \\
& top \rightarrowtail vertex \cdot projCharge \cdot top \\
& top \rightarrowtail vertex \cdot (\wedge_2\ (projAtomType \cdot top)\ (projCharge \cdot top)) \\
& top \rightarrowtail (= 1) \\
& top \rightarrowtail (= 3) \\
& \qquad\qquad \vdots \\
& top \rightarrowtail (= 232) \\
& top \rightarrowtail (\geq -0.781)
\end{aligned}
$$

$$
\begin{aligned}
&(\geq -0.781) \rightarrowtail (\geq -0.424)\\
&\qquad\quad\vdots\\
&(\geq 0.647) \rightarrowtail (\geq 1.004)\\
&top \rightarrowtail (=1)\\
&\qquad\quad\vdots\\
&top \rightarrowtail (=7)\\
&(>0) \rightarrowtail (>1)\\
&\qquad\quad\vdots\\
&(>4) \rightarrowtail (>5).
\end{aligned}
$$

The settings for the learner were $prune = 80\%$, $cutout = 0$, $balance = max(2, 2\%$ of examples), and $splits = 0$. On a 10-fold cross-validation (taking about 40 hours), the average accuracy was 79.06%, and $FP = 0.33$ and $TP = 0.89$. This classifier is the one called 'Edges' in Figure 11. The learner found the following definition on the full data set.

$$
\begin{aligned}
&mutagenic\ m =\\
&\quad if\ edges \cdot (domCard\ (connects \cdot (msetExists_2\ (vertex \cdot projAtomType \cdot (=22))\\
&\qquad (\wedge_2\ (vertex \cdot projAtomType \cdot (=3))\ (vertex \cdot projCharge \cdot (\geq -0.067)))))) \cdot (>4)\ m\\
&\quad then\ \top\\
&\quad else\ if\ edges \cdot (domCard\ (connects \cdot (msetExists_2\\
&\qquad\qquad\qquad\qquad (vertex \cdot projAtomType \cdot (=16))\ top))) \cdot (>0)\ m\\
&\qquad then\ \top\\
&\qquad else\ \bot.
\end{aligned}
$$

"A molecule is mutagenic iff either

(i) it has at least five bonds joining an atom with type 22 and an atom with type 3 and charge ≥ -0.067, or

(ii) it has less than five bonds joining an atom with type 22 and an atom with type 3 and charge ≥ -0.067, and it has an atom of type 16."

It is clear from Figure 11 that the 'Indicators' and 'Indicators and vertices' are optimal classifiers under some conditions since they lie on the edge of the convex hull. Furthermore, the 'Indicators and edges' classifier is so close to the 'Indicators and vertices' classifier that they are essentially equivalent. The other two classifiers lie just off the edge and hence are not optimal under any conditions. However, the 'Vertices' classifier is so close to the edge it is possible it could be useful if a high false positive rate was acceptable.

Perhaps the main conclusion to draw from these mutagenesis experiments is that the approach of specifying the hypothesis language by means of a set of rewrites is flexible and convenient, and gives precise control over the choice of hypothesis language.

Finally, we remark that the accuracy heuristic can be generalised to provide cost-sensitive models. For concreteness, consider Figure 11 again. The models given there on the edge of the convex hull are in the "cost-neutral" region of the ROC diagram in the sense that they are likely to be optimal for iso-performance lines that have gradient close to 1 (that is, the misclassification costs of false positives and false negatives are nearly equal, assuming nearly equal distributions of positive and negative examples). However, sometimes we have a specific requirement for a model in a situation for which the misclassification costs are very different. For example, if $c(\mathbf{Y}, \mathbf{n}) \gg c(\mathbf{N}, \mathbf{p})$, then we are interested in "synthetic-chemist friendly" models that are in the region of the ROC diagram where FP is close to 0 (because in this case the gradient of the iso-performance lines is very steep). Conversely, if $c(\mathbf{Y},\mathbf{n}) \ll c(\mathbf{N},\mathbf{p})$, then we are interested in "toxicologist friendly" models that are in the region of the ROC diagram where TP is close to 1 (because in this case the gradient of the iso-performance lines is very shallow). The cost-sensitive models can be obtained by introducing *relative weights* for the positive and negative examples into the accuracy heuristic – models in the "synthetic-chemist friendly" region can be obtained by a relatively heavier weighting for negative examples, whereas models in the "toxicologist friendly" region can be obtained by a relatively heavier weighting for positive examples. Results for this approach for a predictive toxicology application will be reported elsewhere.

9 Discussion

In this section, we summarise what has been achieved in this paper and place it in the wider context of research into inductive learning.

First, we adopted an "individuals-as-terms" approach to knowledge representation, using a typed, higher-order logic. This approach extends the usual attribute-value approach, which can be understood as restricting the terms to be tuples of constants. We deliberately adopted a rich representation language, which also included sets, multisets, lists, trees and graphs. We then noted that the type selected to model individuals strongly suggests the predicates that will be useful for classifying individuals and used this principle to provide a carefully chosen collection of functions on each type which could be used to construct predicates on terms.

As an illustration of the importance of types, consider the respective uses of sets or lists in knowledge representation. Programming languages do not always support sets well and there is a tendency to use lists for everything. From our perspective, this conflation can be disastrous as the functions associated with sets and lists are very different. These different functions simply reflect the different semantics associated with each of these data types. The likely consequence of using lists when sets should be used, or conversely, is to generate large numbers of inappropriate predicates that are unlikely to lead to the discovery of a good classification function.

Armed with the ability to construct suitable predicates, we then generalised decision-tree learning algorithms to cope with the extra syntax of higher-order terms. This generalisation is theoretically easy enough – it is only necessary to generate suitable predicates for the type of the terms being classified at each node in the decision tree. However, the generalisation does bring a practical difficulty, as in many cases a much larger space of predicates needs to be searched to find a good split than is common in attribute-value learning. For this reason, the search space was carefully structured so that the descendants of a predicate all logically imply the parent predicate. Then, using the accuracy heuristic, we were able to exploit the properties of the refinement bound in Proposition 7.1 to safely prune many predicates. Furthermore, we used a heuristic based on accuracy to help guide the search through the space of predicates. We applied the techniques to ten small-scale applications to illustrate the various types being considered, and also to two larger applications, Musk and Mutagenesis.

During the course of the research that led to this paper, we considered several methods for constructing predicates that were rather different to what is presented here. The earlier methods all had a prominent role for quantified variables and complicated descriptions of the classes of predicates that could be constructed. The biggest problem with these earlier approaches was that it was difficult to comprehend the search space of predicates. The breakthrough to finding something more satisfactory came with the realisation that predicates are compositions of simpler functions and that transformations are sufficiently general to provide all the functions one might want. The result of this approach is that predicates become syntactically simpler; they can be read left to right and all the variables are suppressed. Furthermore, the systematic construction of predicates by using rewrites has two important advantages. The first is that the rewrites provide an easy way to visualise the search space of predicates. Our experience is that, even for complicated applications, the collection of rewrites is compact and the way they generate the search space is comprehensible. The second advantage is that the rewrites give fine control over the predicates constructed – it is easy to experiment with different spaces of predicates, and therefore different hypothesis languages, by appropriately modifying the rewrites. Since constructing predicates is a key component of any symbolic learning system, we believe our higher-order approach is likely to be useful in other subfields of symbolic machine learning.

It is instructive to compare the higher-order approach to machine learning advocated here with the usual first-order approach in the field of inductive logic programming (ILP). Essentially, ILP addresses the same class of problems as is considered in this paper, but by rather different methods. An overview of this first-order approach to learning can be found in [MD94] and an impressive list of applications in [ILPb]. The theory is covered comprehensively in [NCdW97]. There has been a series of ILP workshops held since 1991 [ILPa]. The proceedings of the latest of these, ILP2000, is [CF00]. This comparison is centred around two issues: representation of individuals and predicate (or theory) construction.

First, the representation of individuals is handled differently in each approach, but there is a correspondence between them. In the higher-order case, individuals are represented by basic terms. In contrast, most ILP systems use (untyped) first-order logic for knowledge representation, often with unary type predicates to mimic a type system. In this case, abstractions such as sets and multisets are not directly available and so the usual approach is to name the individual by a constant and then have a relational database of facts about this constant that records the same information as the higher-order term does.

We explore the database versus the term representation in more detail. To illustrate the comparison, what follows is the first-order representation given in [Yor] of the atom d_1 used to illustrate the term representation in Section 3.

$$
\begin{array}{ll}
atm(d_1, d_{1,1}, c, 22, -0.117) & atm(d_1, d_{1,14}, c, 22, -0.117) \\
atm(d_1, d_{1,2}, c, 22, -0.117) & atm(d_1, d_{1,15}, h, 3, 0.143) \\
atm(d_1, d_{1,3}, c, 22, -0.117) & atm(d_1, d_{1,16}, h, 3, 0.143) \\
\vdots & \vdots \\
atm(d_1, d_{1,11}, c, 27, -0.087) & atm(d_1, d_{1,24}, n, 38, 0.812) \\
atm(d_1, d_{1,12}, c, 27, 0.013) & atm(d_1, d_{1,25}, o, 40, -0.388) \\
atm(d_1, d_{1,13}, c, 22, -0.117) & atm(d_1, d_{1,26}, o, 40, -0.388) \\
 & \\
bond(d_1, d_{1,1}, d_{1,2}, 7) & bond(d_1, d_{1,12}, d_{1,20}, 7) \\
bond(d_1, d_{1,1}, d_{1,6}, 7) & bond(d_1, d_{1,13}, d_{1,14}, 7) \\
bond(d_1, d_{1,1}, d_{1,7}, 7) & bond(d_1, d_{1,13}, d_{1,15}, 1) \\
\vdots & \vdots \\
bond(d_1, d_{1,11}, d_{1,12}, 7) & bond(d_1, d_{1,20}, d_{1,23}, 1) \\
bond(d_1, d_{1,11}, d_{1,17}, 7) & bond(d_1, d_{1,24}, d_{1,25}, 2) \\
bond(d_1, d_{1,12}, d_{1,13}, 7) & bond(d_1, d_{1,24}, d_{1,26}, 2).
\end{array}
$$

This first-order representation is called the *flat* representation. Technically, the flat representation is untyped because the underlying logic (and Prolog) is untyped. However, in practice, carefully constructed flat representations are easily typed.

As can be seen, the flat representation is a database of facts about the individual (a molecule, in this case). The molecule is named by the constant d_1 and each of its atoms have names of the form $d_{1,i}$, for some i. Then, essentially, the term used by us is picked apart with a separate fact for each atom and bond. The functions on the various types given above then have counterparts as functions on the flat representation. It is possible to formalise this mapping from the term representation to the flat representation, but we do not pursue this aspect here. (See [Llo01b].)

The situation is summarised in Figure 12 below. On the vertical axis, we have the possible terms used to represent individuals which increase in complexity as one goes up the axis. At the bottom, terms consist just of tuples of constants. Corresponding to the ILP case, terms normally consist of tuples, sets, lists, and constants. There is also the (trivial) mapping from tuples of constants across to the attribute-value language and the more complex mapping from terms to the flat representation used in ILP. This figure also clarifies the connection between attribute-value learning and ILP: reversing the mappings back onto the vertical axis reveals the essential difference is in the complexity of the terms allowed in knowledge representation.

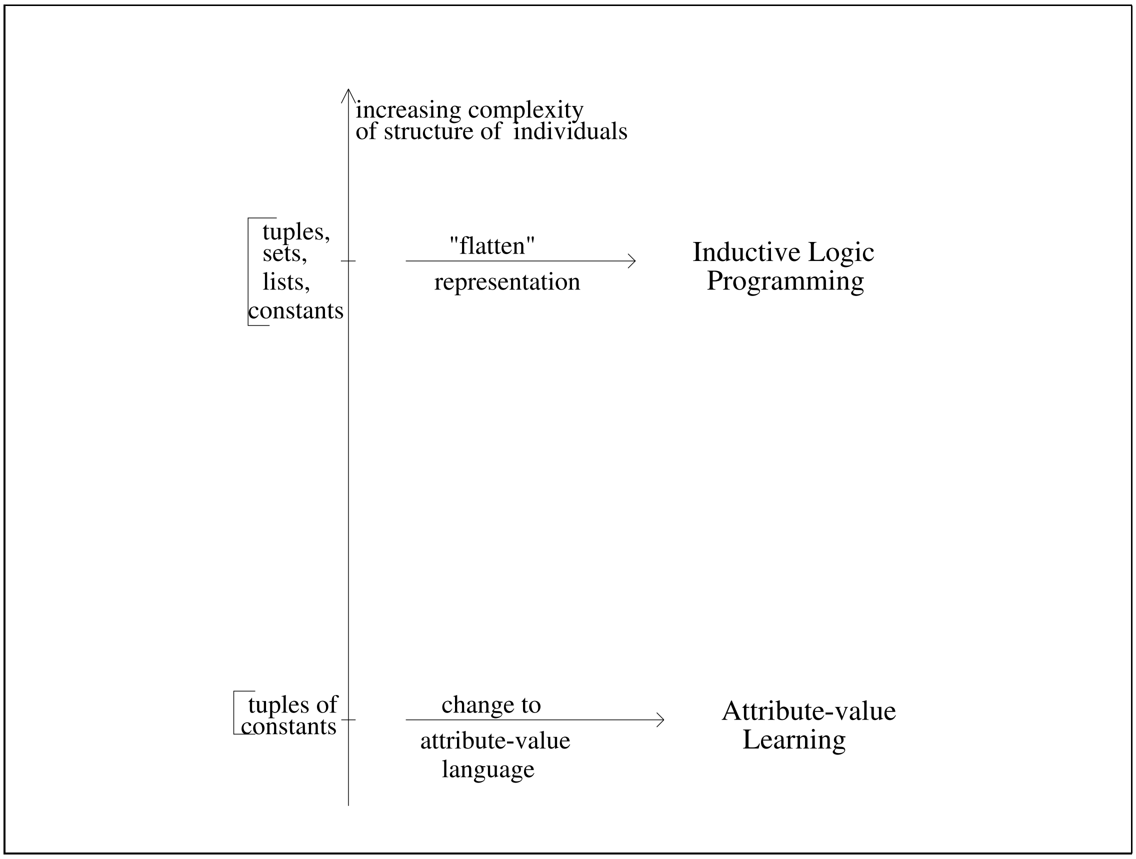

Figure 12: Relationship between attribute-value learning and inductive logic programming based on the individuals-as-terms approach to knowledge representation

So what is there to choose between the term and flat representations? The higher-order approach is more expressive than the first-order one (as we discuss below), is arguably more direct, and corresponds precisely to the principle in logic that (closed) terms denote individuals. However, it is true that all individuals that can be represented as basic terms in higher-order logic can also be represented in first-order logic, so nothing *essential* is lost by staying in first-order logic – given one of these representations and the mapping between them, one can reconstruct the other. The more relevant point is that, since higher-order logic includes all the features of first-order logic and more besides, it is clearly advantageous to work in higher-order logic.

Next we compare the approach to predicate (or theory) construction here and that in ILP. As we shall see, these are different, but there is a correspondence between them. To see this, consider the general statement of the problem of inductive learning for the first-order case, as expounded in [NCdW97, p.167].

> **Given:** A background theory $\mathcal{B}$, a set E^+ of positive examples, and a set E^- of negative examples.
> **Find:** A theory Σ such that $\Sigma \cup \mathcal{B}$ is correct with respect to E^+ and E^-.

Correctness means that $\Sigma \cup \mathcal{B}$ logically implies all the positive examples but none of the negative ones.

Most ILP systems proceed by starting with some initial theory Σ and iteratively specialising $\Sigma \cup \mathcal{B}$, if it is too strong, or generalising $\Sigma \cup \mathcal{B}$, if it is too weak, until $\Sigma \cup \mathcal{B}$ is correct. Typically, both Σ and $\mathcal{B}$ are Horn clause theories and Σ is the definition of some predicate. Such systems are often either top down (that is, only ever specialise) or are bottom up (that is, only ever generalise). A key feature of ILP systems is the use of refinement operators. An upward refinement operator computes generalisations of a clause, while a downward refinement operator computes specialisations of a clause. Much more detail is contained in [NCdW97], but this is enough to be able to draw out the comparison between the first-order and higher-order approaches.

Now we return to the keys example of Subsection 5.2 and Section 6 to draw out the comparison. The induced definition for the function *opens* was found with our decision-tree system, which is a top-down learner. Starting from the most general definition that all bunches of keys open the door, the learner specialises the definition of the predicate until a suitable one is found. The corresponding search tree has the most general predicate *top* at the root of the tree and has successively stronger predicates down the branches of the tree. For this example, the decision-tree has a single split and hence the corresponding definition contains just one *if_then_else* function. More generally, the decision-tree algorithm does a hill-climbing search, at each stage committing to some split of the examples at that node. For each split, there is a corresponding occurrence of the *if_then_else* function in the definition.

To make the comparison of the first-order and higher-order cases clearer, it will be convenient to replace the occurrences of the *if_then_else* function in the definition by occurrences of the if-then-else transformation *ite*. As an example of this, the definition for *opens* can be reformulated as follows.

$$opens = ite\ (setExists_1\ (\wedge_2\ (projMake \cdot (=Abloy))\ (projLength \cdot (=Medium))))\ top\ bottom.$$

More generally, definitions produced by the decision-tree learner will have as many occurrences of *ite* as there are splits in the corresponding decision tree.

With this change, solving the keys problem can be thought of as a top-down search for a definition for *opens* that starts from the most general definition

$$opens = top,$$

and uses the rewrites

$$\begin{aligned}
&top \rightarrowtail ite\ top\ top\ top\\
&top \rightarrowtail setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot top))\\
&top \rightarrowtail (=Abloy)\\
&\qquad\qquad \vdots\\
&top \rightarrowtail (=Broad)\\
&top \rightarrowtail bottom.
\end{aligned}$$

In this formulation, there would need to be careful control of the use of the first rewrite – this control is effectively built into the decision-tree algorithm.

As well as this top-down approach, there are bottom-up ones. In this case, the starting definition for *opens* is the most specific definition

$$opens = bottom,$$

and the rewrites have the form

$$
\begin{aligned}
&\mathit{bottom} \rightarrowtail \mathit{ite\ bottom\ bottom\ bottom}\\
&\mathit{bottom} \rightarrowtail \mathit{setExists_1}\ (\wedge_4\ (\mathit{prMake} \cdot \mathit{bottom})\ (\mathit{prNumProngs} \cdot \mathit{bottom})\ (\mathit{prLength} \cdot \mathit{bottom})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{prWidth} \cdot \mathit{bottom}))\\
&\mathit{bottom} \rightarrowtail (= \mathit{Abloy})\\
&\qquad\quad \vdots\\
&\mathit{bottom} \rightarrowtail (= \mathit{Broad})\\
&\mathit{bottom} \rightarrowtail \mathit{top}.
\end{aligned}
$$

For bottom-up algorithms, the definition of implication-consistent is reversed, so that the head of a rewrite implies the body. With this set of rewrites, the definition for *opens* could be found by an algorithm that successively constructs the following definition bodies.

$$
\begin{aligned}
&\mathit{bottom}\\
&\mathit{ite\ bottom\ bottom\ bottom}\\
&\mathit{ite}\ (\mathit{setExists_1}\ (\wedge_4\ (\mathit{prMake} \cdot \mathit{bottom})\ (\mathit{prNumProngs} \cdot \mathit{bottom})\ (\mathit{prLength} \cdot \mathit{bottom})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{prWidth} \cdot \mathit{bottom})))\ \mathit{bottom\ bottom}\\
&\mathit{ite}\ (\mathit{setExists_1}\ (\wedge_4\ (\mathit{prMake} \cdot (= \mathit{Abloy}))\ (\mathit{prNumProngs} \cdot \mathit{bottom})\ (\mathit{prLength} \cdot \mathit{bottom})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{prWidth} \cdot \mathit{bottom})))\ \mathit{bottom\ bottom}\\
&\qquad\quad \vdots\\
&\mathit{ite}\ (\mathit{setExists}_1\ (\wedge_4\ (\mathit{prMake} \cdot (= \mathit{Abloy}))\ (\mathit{prNumProngs} \cdot \mathit{top})\ (\mathit{prLength} \cdot (= \mathit{Medium}))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{prWidth} \cdot \mathit{top})))\ \mathit{top\ bottom}.
\end{aligned}
$$

The comparison between the first-order approach and higher-order approach to predicate (or theory) construction should be clear by now. In the top-down case, the definition body *top* corresponds to the most general theory and applying a rewrite corresponds to applying a downward refinement operator. In the bottom-up case, the definition body *bottom* corresponds to the most specific theory and applying a rewrite corresponds to applying an upward refinement operator.

To some extent, whether one uses the first-order approach or the higher-order approach to predicate construction (for first-order theories) is a matter of taste – whatever can be done in the higher-order setting can be done in the first-order one, possibly by rather indirect methods. Nevertheless, there are advantages in using the higher-order setting. First, in addition to all the first-order facilities, there are the strictly higher-order ones that prove to be very difficult to do without once one gets used to their convenience and expressive power. Second, as we have documented in this paper, the technique of specifying the hypothesis language by means of rewrites has proved to be highly successful for decision-tree learning. Constructing theories by specifying a set of rewrites gives the user precise and explicit control over the search space of predicates. Furthermore, this method banishes troublesome variables altogether from the set of rewrites and the predicates constructed – variables are needed to define the transformations, but after that they do not appear. This completes the comparison.

We now turn to specific related work. A general discussion of the influence of knowledge representation on the computational complexity of learning, including a useful list of references on this topic up to 1994, is contained in [NS94]. An earlier paper, which contains a discussion of the possible choices available for controlling complexity in first-order learning and the particular choices made for the RDT system, is [KW92]. Section 7 of [MD94] contains a discussion of declarative bias in ILP. Declarative bias is divided up into syntactic bias (also called language bias) and semantic bias, where syntactic bias imposes restriction on the form of clauses allowed in hypotheses and semantic bias is said to include types and modes. (The inclusion of types as a *semantic* bias is clearly not appropriate for a typed logic.) The discussion in [MD94] covers the main approaches to specifying declarative bias up to 1994. These include [Coh94] in which a first-order learning system that provides a language for explicitly stating language bias is described. This paper is particularly interesting and has several parallels with the approach here, so we discuss it in some detail.

The main technical innovation in [Coh94] is the use of so-called antecedent description grammars that serve a similar purpose for the first-order case as the rewrite rules introduced in this paper serve for the higher-order case. The simplest way of explaining the connection between antecedent description grammars and higher-order rewrite rules is to give an example. We use the running example about the *illegal* chess problem, that is, the problem of learning when a chess position containing two kings and one rook is illegal. The following description is taken from [Coh94, p.306-307]. The problem is that of learning the predicate $illegal(A, B, C, D, E, F)$, where A and B are the rank and file of the white king, C and D are the rank and file of the white rook, and E and F are the rank and file of the black king. A position is *illegal* if the two kings are adjacent, if two pieces occupy the same square, or if the black king is in check. (We assume it is white's move.) The predicates available for inclusion in the hypothesis are $adj(X, Y)$, which is true if the ranks or files X and Y are adjacent, $less\_than(X, Y)$, which is true if rank or file X precedes Y, and $equal(X, Y)$.

A grammar defining one possible antecedent description language for this learning problem is as follows.

$$goal\_formula(illegal(A, B, C, D, E, F)).$$

$$body(illegal(A, B, C, D, E, F)) \to rels(A, B, C, D, E, F).$$
$$rels(A, B, C, D, E, F) \to [].$$
$$rels(A, B, C, D, E, F) \to rel(A, B, C, D, E, F), rels(A, B, C, D, E, F).$$

$$rel(A, B, C, D, E, F) \to pred(X, Y)$$
$$\quad where\ member(X, [A, B, C, D, E, F]), member(Y, [A, B, C, D, E, F]).$$

$$pred(X, Y) \to [equal(X, Y)].$$
$$pred(X, Y) \to [\neg equal(X, Y)].$$
$$pred(X, Y) \to [adj(X, Y)].$$
$$pred(X, Y) \to [\neg adj(X, Y)].$$
$$pred(X, Y) \to [less\_than(X, Y)].$$
$$pred(X, Y) \to [\neg less\_than(X, Y)].$$

The first line of the grammar declares which predicate is the *goal formula*, that is, it defines the name of the concept to be learned. If the goal formula is G, then $body(G)$ is the designated start symbol of the grammar. The terminal symbols are enclosed in square brackets, and [] denotes the empty string. A rule of the form

$$A \to B\ where\ P$$

is a shorthand for all the rules that can be obtained from this one by applying the various substitutions that are solutions of the Prolog goal P. Thus the rule above having the *where* clause is a shorthand for the following set of 36 rules.

$$rel(A, B, C, D, E, F) \to pred(A, A).$$
$$rel(A, B, C, D, E, F) \to pred(A, B).$$
$$\vdots$$
$$rel(A, B, C, D, E, F) \to pred(F, F).$$

These clauses are generated by a macro-expansion facility when the grammar is read in. Thus the grammar defines clauses that have $illegal(A, B, C, D, E, F)$ in the head and have bodies that are conjuncts of literals such as $adj(A, E)$ and $\neg equal(C, E)$. It is shown in [Coh94] that antecedent description grammars can be used to make use of many different kinds of background knowledge, including constraints on how predicates can be used, programming clichés, overgeneral theories, incomplete theories, and theories syntactically close to the target theory.

We now show how the above grammar would be represented as a rewrite system. Let *Coord* be the type of a rank or file and *Position* be the type of 6-tuples, each of whose arguments has type *Coord*.

$$Position = Coord \times \cdots \times Coord.$$

Then the function we want to learn is

$$illegal : Position \rightarrow \Omega.$$

The following transformations will be needed.

$$\begin{aligned}
&\wedge_2 : (Position \rightarrow \Omega) \rightarrow (Position \rightarrow \Omega) \rightarrow Position \rightarrow \Omega \\
&proj_{i,j} : Position \rightarrow Coord \times Coord \\
&equal : Coord \times Coord \rightarrow \Omega \\
&adj : Coord \times Coord \rightarrow \Omega \\
&less_than : Coord \times Coord \rightarrow \Omega.
\end{aligned}$$

The transformation $proj_{i,j}$ is defined by $proj_{i,j}\ (x_1, \ldots, x_6) = (x_i, x_j)$, for $i, j = 1, \ldots, 6$. The rewrites are as follows.

$$\begin{aligned}
&top \rightarrowtail \wedge_2\ top\ top \\
&top \rightarrowtail proj_{1,1} \cdot top \\
&\qquad\quad \vdots \\
&top \rightarrowtail proj_{6,6} \cdot top \\
&top \rightarrowtail equal \\
&\qquad\quad \vdots \\
&top \rightarrowtail less_than \cdot \neg.
\end{aligned}$$

This set of rewrites produces exactly the same set of bodies for the definition of *illegal* as does the antecedent description grammar. Disjunction can easily be added to the rewrite set to produce function bodies that correspond to multi-clausal theories.

The conclusion that can be drawn from this comparison is that, for the first-order case, antecedent description grammars and rewrite systems have much the same expressive power. The biggest difference is the lack of a macro-expansion facility for rewrites. In the applications we considered in this paper, we never felt the need for such a facility, although the systems of rewrites for the Musk application would benefit from macro-expansion. In any case, it's clear that it could be added easily, if desired. For example, the 36 rewrites above involving the projections could be written more compactly as

$$top \rightarrowtail proj_{i,j} \cdot top \text{ where } i \in \{1, 2, 3, 4, 5, 6\} \wedge j \in \{1, 2, 3, 4, 5, 6\}.$$

Examination of the subsequent variant grammars for *illegal* and the other applications in [Coh94] supports the claim that the grammars there can also be written straightforwardly as rewrites. This implies that the advantages for grammars listed above also apply to rewrites, with rewrites having the further advantage that they can also be used in higher-order contexts. We plan to investigate further the applications considered in [Coh94] (which happen to be disjoint from the applications considered here) to see what further parallels there are and also to try to formalise the precise relationship between the two approaches.

One paper that advocates the usefulness of sets in machine learning is [Coh96]. 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, [Coh96] makes a strong case for the practicality of using set-valued features.

In [CH94], the learnability of description logics is studied. Description logics have some similarities with higher-order logics; in particular, they admit so-called constructors, such as `AND`, `ALL`, and `AT-LEAST`, which have similar semantics to expressions provided by higher-order logic. Description logics are primarily used for taxonomic reasoning and hence an important operation is determining if one description is more general than another. Since individuals are atomic in description logics, for learning purposes, it is necessary to deal with descriptions. In this setting, examples are descriptions and an example is marked as positive if it is subsumed by the target concept. A comparison of learning in the description logic framework with ILP is provided in [CH94] and the conclusion is drawn that the knowledge representations are so different that a direct comparison is difficult. A similar comment applies to the comparison with the higher-order framework presented here.

A paper advocating the utility of higher-order logic for learning is [MP94], but the work there is seriously limited by its reliance on Prolog's inadequate higher-order features. Part of the motivation of this paper is to show that the intuition in [MP94] is correct, but that one needs to work in a typed higher-order logic to achieve its aims.

Two papers that make use of functions in learning are [FRHORQ00] and [FRHORQ01]. The formalism underlying the work there is a functional logic programming system called FLIP that provides a general-purpose learning system, useful for diverse tasks such as learning recursive definitions of functions and data mining. There is no obvious use of types or higher-order features in these papers, but the motivation for introducing functions and equality is the same as ours.

A recent learning system closely related to ours is the Tilde system, described in [BD97] and [BD98]. 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 differences. First, Tilde uses the flat representation of Prolog, as is common in ILP systems. Second, it allows variables and conditions to be spread across several nodes of the tree which causes some complications. Prior to Tilde there were several systems that extended decision-tree learning to the first-order case. These include the structural decision trees in [WR91] and the structural regression trees in [Kra96]. Each of these systems uses the flat representation; they also construct conditions that are confined to a single node for splitting examples, as we do. A top-down learning algorithm is given in [Bos95]. This system neatly induces programs in a top-down fashion by the use of the unfold program transformation to specialise clauses. A number of authors, for example, [BFOS84], have previously suggested using conjunctions of conditions to split a node in decision-tree algorithms.

Following this discussion of previous work, the contribution of this paper can be put into perspective. Early, and much of the present, work in machine learning is set in the context of the attribute-value language, that is, individuals are represented by feature tuples of numbers or constants. Even in the case when the individuals have complex structure, it is often possible to get good results by choosing a collection of features to represent the individuals. But this approach clearly has its limitations and so researchers were motivated to set their work in more expressive languages, particularly, first-order logic and the language of Horn clauses. This approach goes back to the pioneering work of Plotkin [Plo70], Shapiro's MIS system [Sha83], and Michalski's foundational work on inductive learning [Mic83]. Around ten years ago, there was a substantial increase in interest in more expressive knowledge representation formalisms with the establishment of the subfield of inductive logic programming led by Muggleton and De Raedt [MD94]. Much of the important work on learning in first-order logic has taken place in ILP and most of the standard techniques of machine learning have now been upgraded to this context.

So where might we go from here? An obvious idea is to upgrade machine learning techniques so that they use even more expressive knowledge representation languages. While there are a number of potential choices, including modal and temporal logics, an obvious candidate is higher-order logic. There are at least two good reasons for this.

The first reason is that suitable programming languages are needed to support the learning mechanisms (as Prolog does for ILP) and there now exist excellent higher-order declarative programming languages, including Haskell [Je], λ-Prolog [NM88], and Curry [He]. In particular, Curry looks to be highly suitable for learning applications since it is a functional logic programming language with all features of Haskell together with the logic programming capabilities of Prolog. A notable feature of Haskell and Curry (and [Llo99]) is that programs are *equational* theories rather than implicational ones as for Prolog. The reason is that equality is more flexible than implication since its arguments can have any type. (Equality is also simpler to understand: it's easier to comprehend, in the boolean case, that two items are identical rather than that one implies the other.) This use of equality means that programs admit definitions of functions rather than just predicates. Ultimately it's tiresome being restricted to clausal theories and predicate definitions – one of advantages of the approach adopted here is the flexibility of function definitions (whether higher-order or not).

Another reason for moving to higher-order logic is that it has greater expressive power than first-order logic. First, the logic has a powerful type system that we have exploited throughout the paper. (First-order logic can also be typed, of course, but this has not been fully exploited in its applications to machine learning.) Second, higher-order logic allows abstractions such as (finite) sets and multisets to appear in terms representing individuals. The expressive power provided by basic terms with their combination of structures, abstractions, and tuples is greater than that provided by first-order terms. As we pointed out above, this does not mean that first-order logic cannot represent the same class of individuals by other, less direct, means. However, it is worth pointing out that in first-order logic, even something as simple as a tuple is *not* a term – 'tuples' can only exist as arguments to functions or predicates. Furthermore, while it is true that one can axiomatise set theory in first order logic, it would be unthinkable to use such an impractical method in a computation system. Also, *extensional* sets can be admitted by adding certain data constructors with special equational theories, but this does not allow *intensional* sets. In any case, none of these first-order techniques cover the full generality of basic abstractions, except by mapping to the flat representation. In contrast, *all* the first-order representation techniques are *directly* available in higher-order logic. The third reason that higher-order logic is more expressive than first-order logic is that it admits higher-order functions, that is, functions that can have other functions as arguments. We exploit this feature heavily in our construction of predicates from transformations. Once again, this does not mean that first-order logic cannot be used to construct predicates – of course, ILP systems do this all the time. But the predicate construction method shown here, for example, is inaccessible to first-order logic (because it cannot even be directly *formulated* in first-order logic), whereas *all* the first-order methods are *directly* available in higher-order logic. As another example, higher-order logic is needed just to formulate the problem of learning definitions in higher-order programming languages, such as Haskell or Curry, much less solve it. In summary, nothing at all is lost by moving to higher-order logic and there is clearly a lot to gain.

This move to more expressive formalisms parallels the move from assembly languages to high-level programming languages that allow programmers to write their programs using notations and concepts closer to those used by humans. In other parts of computer science (for example, in theoretical computer science, functional programming, and hardware verification), the move to higher-order logic took place a long time ago and these areas now depend fundamentally on its higher-order nature. If the history of declarative programming languages (a key technology in machine learning) is a guide, then the upgrade of machine learning techniques to higher-order logic is inevitable. Furthermore, as the history of science shows, moving to a more expressive formalism is often the key to further breakthroughs.

This paper has attempted to lay some foundations, especially concerning knowledge representation, for machine learning in higher-order logic. In spite of its length, it leaves many questions unanswered. Nevertheless we hope that our detailed analysis of knowledge representation and predicate construction, and our experiments with the decision-tree learning system, provide sufficient evidence to suggest that upgrading from the attribute-value formalism and first-order logic to higher-order logic will be beneficial.

We conclude with some remarks about future research directions. In this paper we have concentrated on the problem of classification in inductive learning, particularly for decision-tree learning. The main challenges for the future are concerned with the application of the higher-order approach to knowledge representation and predicate construction to other subfields of machine learning. First, the ideas here have been applied successfully to evolutionary learning in [KGC99a] and [KGC99b], and it would be interesting to develop these further. They also allow one to generalise neural networks from ones that handle individuals represented by tuples of constants to ones that directly handle individuals represented by basic terms – this work will be reported elsewhere. Modifying the decision-tree system to perform regression rather than classification is clearly straightforward, but would have many interesting applications.

One particularly interesting possibility is that of learning the definitions of higher-order functions of a programming language such as Haskell. Because of the huge potential for applications in software engineering, there has been a lot of work to find ways of learning Horn clause programs, especially ones involving recursive predicates. The recursive nature of some predicates is certainly a stumbling block to further progress. However, there is a possible approach to avoiding the difficulties of recursion that intrinsically depends on the higher-order nature of the logic. Many Haskell functions, whose corresponding programs in Prolog are written recursively, are usually written without *explicit* recursion by using functions such as `foldr` that package up recursion into a convenient form. This strongly suggests that the definitions of such functions can be learned with a decision-tree system in the following way. First, convert, if necessary, the signature of function to the signature of its corresponding predicate (that is, $f : \sigma \to \tau$ is converted to $p : \sigma \times \tau \to \Omega$) , with a similar modification to the examples. (We propose to learn the definition of the predicate, because its codomain is the booleans and can be handled by a classification system, and then extract the required definition of the function from that of the predicate.) Then add a small number of higher-order functions, such as `foldr`, to the background theory. Next a suitable set of rewrites needs to be found. This is the part that requires further research as what is needed is a reasonably comprehensive set of rewrites that provides enough patterns, involving `foldr` etc., to cover the likely uses of recursion and does not need programmer intervention. (To provide the appropriate level of automation for learning function definitions for programming, the system must have a suitable set of rewrites already available that does not require programmer intervention.) The next step is to run the decision-tree learner to induce the definition of the predicate. Finally, the function definition is extracted from that of the predicate. If this approach works, it will certainly provide a strong motivation for moving to higher-order logic.

As mentioned in the introduction, the set of basic terms can be made into a metric space in a natural way [Llo01a, Llo01b]. This metric provides a way of generalising both nearest neighbour classification algorithms and clustering algorithms to individuals represented by basic terms. Earlier in this section, we indicated that our approach to predicate construction also allows bottom-up construction, so it would be interesting to develop learning algorithms based on these ideas. The existing approaches to Bayesian networks [Pea88] and reinforcement learning [SB98] should also benefit from the extra expressiveness of higher-order logic as a representation language. (For a first-order treatment of reinforcement learning, see [DDB98].) Another promising idea is to link up the approach of this paper and well established kernel methods. To make this work, mappings must be defined (at least implicitly) from the spaces of individuals considered here to (possibly high-dimensional) feature spaces of the form $\mathbb{R}^n$, where $\mathbb{R}$ is the real numbers. In fact, it suffices to define an "inner product" on pairs of individuals that implicitly defines such a mapping and then apply linear optimisation methods, as support vector machines do [SBS98].

10 Conclusion

The research in this paper addressed the issue of knowledge representation in inductive learning in a novel way. The basic principles employed here were that

1. an individual should be represented by a (closed) term, and

2. the type selected to model individuals strongly suggests the predicates that will be useful for classifying individuals.

We demonstrated how the systematic application of these principles provides a framework for classifying individuals having complex structure. We believe that the general approach to knowledge representation and predicate construction proposed here will also be useful in other areas of machine learning.

Acknowledgements

We are greatly indebted to Kee Siong Ng, who implemented the decision-tree system that was used to obtain the main results of this paper. He also carried out the Mutagenesis experiments and, on an earlier implementation, the experiments for the ten easy pieces. Xiao-bing Wu carried out the Musk experiments. Thanks are due to Torbjørn Dahl, Peter Flach, Claire Kennedy, Nicolas Lachiche, and René MacKinney-Romero, current or former members of the Machine Learning Research Group at Bristol University, for many interesting discussions concerning the topic of this paper. Particular thanks are due to Peter Flach, who helped clarify many issues. We thank Claude Sammut, Ashwin Srinivasan, and Rodney Topor for comments on an earlier version of this paper and helpful discussions. Discussions with Hendrik Blockeel, William Cohen, Luc De Raedt, Nada Lavrac, Stephen Muggleton, David Page, and Ross Quinlan were influential in developing the ideas in the paper. An earlier version of the decision-tree system, based on an implementation of the Escher programming language, was written by Antony Bowers. The Escher abstract machine was designed by Kerstin Eder and Ian Holyer. The Escher system was implemented by Antony Bowers, Kerstin Eder, and Ian Holyer. This work was partially supported by EPSRC Grant GR/L21884 and the Australian National University.

References

[BD97] H. Blockeel and L. De Raedt. Top-down induction of logical decision trees. Available at `http://www.cs.kuleuven.ac.be/~hendrik`, 1997.

[BD98] H. Blockeel and L. De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101:285–297, 1998.

[BFOS84] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees.* Chapman and Hall, 1984.

[BGCL00] A.F. Bowers, C. Giraud-Carrier, and J.W. Lloyd. Classification of individuals with complex structure. In P. Langley, editor, *Machine Learning: Proceedings of the Seventeenth International Conference (ICML2000)*, pages 81–88. Morgan Kaufmann, 2000.

[Bon70] M. Bongard. *Pattern Recognition.* Spartan Books, 1970.

[Bos95] H. Boström. Covering vs. divide-and-conquer for top-down induction of logic programs. In *Proceedings of 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1194–1200. Morgan Kaufmann, 1995.

[CF00] J. Cussens and A. Frisch, editors. *Proceedings of the 10th International Workshop on Inductive Logic Programming, ILP2000*, Lecture Notes in Artificial Intelligence, 1866. Springer-Verlag, 2000.

[CH94] W.W. Cohen and H. Hirsh. Learning the CLASSIC description logic: Theoretical and experimental results. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, pages 121–133. Morgan Kaufmann, 1994.

[Coh94] W.W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, 1994.

[Coh96] W.W. Cohen. Learning trees and rules with set-valued features. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 709–716, Menlo Park, CA, 1996. AAAI Press.

[DDB98] S. Džeroski, L. De Raedt, and H. Blockeel. Relational reinforcement learning. In *Proceedings of the 15th International Conference on Machine Learning, ICML'98*, pages 136–143. Morgan Kaufmann, 1998.

[DLLP97] T.G. Dietterich, R.H. Lathrop, and T. Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89:31–71, 1997.

[Eas] Home page of East-West Challenge. `http://www.gmd.de/ml-archive/ILP/public/data/east_west/`.

[FGCL98] P. Flach, C. Giraud-Carrier, and J.W. Lloyd. Strongly typed inductive concept learning. In *Proceedings of the International Conference on Inductive Logic Programming, ILP'98*, pages 185–194, 1998.

[FMPS98] P. Finn, S. Muggleton, D. Page, and A. Srinivasan. Pharmacophore discovery using the inductive logic programming system PROGOL. *Machine Learning*, 30:241–270, 1998.

[FRHORQ00] C. Ferri-Ramírez, J. Hernández-Orallo, and M.J. Ramírez-Quintana. Learning functional logic classification concepts from databases. In *Proceedings of the 9th International Workshop on Functional and Logic Programming (WFLP 2000)*, pages 296–308. UPV University Press, Valencia, 2000/2039, 2000.

[FRHORQ01] C. Ferri-Ramírez, J. Hernández-Orallo, and M.J. Ramírez-Quintana. Incremental learning of functional logic programs. In *Proceedings of the 5th International Symposium on Functional and Logic Programming (FLOPS 2001)*, pages 233–247. Springer, 2001. Lectures Notes in Computer Science 2024.

[He] M. Hanus (ed.). Curry: An integrated functional logic language. Available at `http://www.informatik.uni-kiel.de/~curry`.

[ILPa] Home page of ILP workshops. `http://www.cs.york.ac.uk/ILP-events/`.

[ILPb] Home page of ILPNET. `http://www-ai.ijs.si/ilpnet.html/`.

[Je] S. Peyton Jones and J. Hughes (editors). Haskell98: A non-strict purely functional language. Available at `http://haskell.org/`.

[KGC99a] C.J. Kennedy and C. Giraud-Carrier. An evolutionary approach to concept learning with structured data. In *Proceedings of the Fourth International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA'99)*, pages 331–366. Springer-Verlag, 1999.

[KGC99b] C.J. Kennedy and C. Giraud-Carrier. Predicting chemical carcinogenesis using structural information only. In *Proceedings of the Third European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDDD'99)*, pages 360–365, 1999.

[KMSS96] R. King, S. Muggleton, S. Srinivasan, and M. Sternberg. 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, 1996.

[Kra96] S. Kramer. Sructural regression trees. In *Proceedings of 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 812–819. AAAI Press/MIT Press, 1996.

[KW92] J.-U. Kietz and S. Wrobel. Controlling the complexity of learning in logic through syntactic and task-oriented models. In S. Muggleton, editor, *Inductive Logic Programming*, chapter 16, pages 335–359. Academic Press, 1992.

[Llo99] J.W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3), March 1999.

[Llo01a] J.W. Lloyd. Higher-order computational logic. In A. Kakas and F. Sadri, editors, *Computational Logic: From Logic Programming into the Future*. Springer-Verlag, 2001. Volume in honour of the 60th birthday of R.A. Kowalski, in press.

[Llo01b] J.W. Lloyd. Knowledge representation, computation, and learning in higher-order logic. Available at `http://csl.anu.edu.au/~jwl`, 2001.

[LM98a] H. Liu and H. Motoda, editors. *Feature Extraction, Construction and Selection: A Data Mining Perspective*. Kluwer, 1998.

[LM98b] H. Liu and H. Motoda. *Feature Selection for Knowledge Discovery and Data Mining*. Kluwer, 1998.

[MD94] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[Mic83] R.S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20:111–116, 1983.

[Mit82] T.M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

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

[ML77] R.S. Michalski and J.B. Larson. Inductive inference of VL decision rules. In *Workshop on Pattern-directed Inference Systems*, pages 34–44, 1977.

[MP94] S. Muggleton and C.D. Page. Beyond first-order learning: Inductive learning with higher order logic. Technical Report PRG-TR-13-94, Oxford University Computing Laboratory, 1994.

[NCdW97] S.H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Lecture Notes In Artificial Intelligence, 1228. Springer-Verlag, 1997.

[NM88] G. Nadathur and D. Miller. An overview of λProlog. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming,* Seattle, pages 810–827. MIT Press, 1988.

[NS94] F. Neri and L. Saitta. Knowledge representation in machine learning. In F. Bergadano and L. De Raedt, editors, *Proceedings of the European Conference on Machine Learning*, pages 20–27. Springer-Verlag, LNAI 784, 1994.

[Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[PF01] F. Provost and T. Fawcett. Robust classification for imprecise environments. *Machine Learning*, 42(3):203–231, 2001.

[Plo70] G.D. Plotkin. A note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163, 1970.

[Qui86] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Qui93] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[Qui96] J.R. Quinlan. Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, 5:139–161, 1996.

[Sam81] C. Sammut. *Learning Concepts by Performing Experiments*. PhD thesis, University of South Wales, Australia, 1981.

[SB98] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[SBS98] B. Schölkopf, C. Burges, and A. Smola, editors. *Advances in Kernel Methods: Support Vector Learning*. MIT press, 1998.

[Sha83] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[SKM99] A. Srinivasan, R.D. King, and S.H. Muggleton. The role of background knowledge: using a problem from chemistry to examine the performance of an ILP program. Available at `http://www.comlab.ox.ac.uk/oucl/groups/machlearn/`, 1999.

[SMKS94] A. Srinivasan, S. Muggleton, R. King, and M. Sternberg. Mutagenesis: ILP experiments in a non-determinate biological domain. In S. Wrobel, editor, *Proceedings of Fourth Inductive Logic Programming Workshop*. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994. GMD-Studien Nr 237.

[WR91] L. Watanabe and L. Rendell. Learning structural decision trees from examples. In *Proceedings of 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 770 – 776. Morgan Kaufmann, 1991.

[Yor] Home page of Machine Learning Group, The University of York. `http://www.cs.york.ac.uk/mlg/`.

A Appendix: Aspects of Higher-order Logic

The main aim of this appendix is to give the definitions of basic terms, which are used for knowledge representation, and regular predicates, which are produced by predicate construction. Each of these definitions is given in the most appropriate setting, which is that of higher-order logic. Further details are contained in [Llo01a] and [Llo01b].

A.1 Types

Definition. An *alphabet* consists of four sets:

1. A set $\mathfrak{T}$ of type constructors.
2. A set $\mathfrak{P}$ of parameters.
3. A set $\mathfrak{C}$ of constants.
4. A set $\mathfrak{V}$ of variables.

Each type constructor in $\mathfrak{T}$ has an arity. The set $\mathfrak{T}$ always includes the type constructors $\mathit{1}$ and Ω both of arity 0. $\mathit{1}$ is the type of some distinguished singleton set and Ω is the type of the booleans. The set $\mathfrak{P}$ is denumerable (that is, countably infinite). Parameters are type variables and are typically denoted by a, b, c, ... Each constant in $\mathfrak{C}$ has a signature (see below). The set $\mathfrak{V}$ is also denumerable. Variables are typically denoted by x, y, z, ... For any particular application, the alphabet is assumed fixed and all definitions are relative to the alphabet.

Types are built up from the set of type constructors and the set of parameters, using the symbols $\rightarrow$ and $\times$.

Definition. A *type* is defined inductively as follows.

1. Each parameter in $\mathfrak{P}$ is a type.
2. If T is a type constructor in $\mathfrak{T}$ of arity k and $\alpha_1, \ldots, \alpha_k$ are types, then $T\ \alpha_1 \ldots \alpha_k$ is a type. (For $k = 0$, this reduces to a type constructor of arity 0 being a type.)
3. If α and β are types, then $\alpha \rightarrow \beta$ is a type.
4. If $\alpha_1, \ldots, \alpha_n$ are types, then $\alpha_1 \times \cdots \times \alpha_n$ is a type. (For $n = 0$, this reduces to $\mathit{1}$ being a type.)

$\mathfrak{S}$ denotes the set of all types obtained from an alphabet ($\mathfrak{S}$ for 'sort'). The symbol $\rightarrow$ is right associative, so that $\alpha \rightarrow \beta \rightarrow \gamma$ means $\alpha \rightarrow (\beta \rightarrow \gamma)$.

Definition. A type is *closed* if it contains no parameters.

Notation. $\mathfrak{S}^c$ denotes the set of all closed types obtained from an alphabet.

Note that $\mathfrak{S}^c$ is non-empty, since $\mathit{1}, \Omega \in \mathfrak{S}^c$.

Example. In practical applications of the logic, a variety of types is needed, including $\mathit{1}$, Ω, *Nat* (the type of natural numbers), *Int* (the type of integers), *Float* (the type of floating-point numbers), *Char* (the type of characters), and *String* (the type of strings). Each of these is a nullary type constructor.

Other useful type constructors are those used to define lists, trees, and so on. In the logic, *List* denotes the (unary) list type constructor. Thus, if α is a type, then *List* α is the type of lists whose elements have type α.

Use will be made of the concept of one type being more general than another.

Definition. Let α and β be types. Then α is *more general than* β if there exists a type substitution ξ such that $\beta = \alpha\xi$.

Note that "more general than" includes "equal to", since ξ can be the identity substitution.

Example. Let $\alpha = (List\ a) \times \Omega$ and $\beta = (List\ Int) \times \Omega$. Then α is more general than β, since $\beta = \alpha\xi$, where $\xi = \{a/Int\}$.

A.2 Terms

Definition. A *signature* is the declared type for a constant.

Notation. The fact that a constant C has signature α is sometimes denoted by $C : \alpha$.

We distinguish two different kinds of constants: *data constructors* and *functions*. In a knowledge representation context, data constructors are used to represent individuals. In a programming language context, data constructors are used to construct data values. In contrast, functions are used to compute on data values; functions have definitions while data constructors do not. In the semantics for the logic, the data constructors are used to construct models. As examples, the constants $\top$ (true) and $\bot$ (false) are data constructors, as is each integer, floating-point number, and character. The constant : (cons) used to construct lists is a data constructor.

The set $\mathfrak{C}$ always includes the following constants (where a is a parameter).

1. $()$, having signature $\mathit{1}$.
2. $=$, having signature $a \rightarrow a \rightarrow \Omega$.
3. $\top$ and $\bot$, having signature Ω.
4. $\neg$, having signature $\Omega \rightarrow \Omega$.
5. $\wedge$, $\vee$, $\longrightarrow$, $\longleftarrow$, and $\longleftrightarrow$, having signature $\Omega \rightarrow \Omega \rightarrow \Omega$.
6. Σ and Π, having signature $(a \rightarrow \Omega) \rightarrow \Omega$.

The intended meaning of $=$ is identity (that is, $=\ x\ y$ is $\top$ iff x and y are identical), the intended meaning of $\top$ is true, the intended meaning of $\bot$ is false, and the intended meanings of the connectives $\neg$, $\wedge$, $\vee$, $\longrightarrow$, $\longleftarrow$, and $\longleftrightarrow$ are as usual. The intended meaning of Σ is that it is true iff its argument is a predicate that is true on *some* domain element, and the intended meaning of Π is that it is true iff its argument is a predicate that is true on *every* domain element.

Data constructors always have a signature of the form $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow (T\ a_1 \ldots a_k)$, where T is a type constructor of arity k, $a_1, \ldots, a_k$ are distinct parameters, and all the parameters appearing in $\sigma_1, \ldots, \sigma_n$ occur among $a_1, \ldots, a_k$ ($n \geq 0$, $k \geq 0$). Furthermore, for each type constructor T, we assume that there does exist at least one data constructor having a signature of the form $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow (T\ a_1 \ldots a_k)$.

Example. The data constructors for constructing lists are [] having signature $List\ a$ and : having signature $a \rightarrow List\ a \rightarrow List\ a$, where : is usually written infix. [] represents the empty list. The term $s : t$ represents the list with head s and tail t. Thus $4 : 5 : 6 : []$ represents the list $[4, 5, 6]$.

The next task is to define the central concept of a term. In the non-polymorphic case, a simple inductive definition suffices. But the polymorphic case is more complicated since, when putting terms together to make larger terms, it is generally necessary to solve a system of equations and these equations depend upon the relative types of free variables in the component terms. The effect of this is that to define a term one has to define simultaneously its type, and its set of free variables and their relative types.

Definition. A *term*, together with its type, and its set of free variables and their relative types, is defined inductively as follows.

1. Each variable x in $\mathfrak{V}$ is a term of type a, where a is a parameter.

   The variable x is free with relative type a in x.

2. Each constant C in $\mathfrak{C}$, where C has signature α, is a term of type α.

3. (Abstraction) If t is a term of type β and x a variable in $\mathfrak{V}$, then $\lambda x.t$ is a term of type $\alpha \to \beta$, if x is free with relative type α in t, or type $a \to \beta$, where a is a new parameter, otherwise.

   A variable other than x is free with relative type σ in $\lambda x.t$ if the variable is free with relative type σ in t.

4. (Application) If s is a term of type $\alpha \to \beta$ and t a term of type γ such that the equation
   $\alpha = \gamma$,
   augmented with equations of the form
   $\rho = \delta$,
   for each variable that is free with relative type ρ in s and is also free with relative type δ in t, have a most general unifier θ, then $(s\ t)$ is a term of type $\beta\theta$.

   A variable is free with relative type $\sigma\theta$ in $(s\ t)$ if the variable is free with relative type σ in s or t.

5. (Tupling) If $t_1, \ldots, t_n$ are terms of type $\alpha_1, \ldots, \alpha_n$, respectively, such that the set of equations of the form
   $\rho_{i_1} = \rho_{i_2} = \ldots = \rho_{i_k}$,
   for each variable that is free with relative type ρ_{i_j} in the term t_{i_j} ($j = 1, \ldots, k$ and $k > 1$), have a most general unifier θ, then $(t_1, \ldots, t_n)$ is a term of type $\alpha_1\theta \times \cdots \times \alpha_n\theta$.

   A variable is free with relative type $\sigma\theta$ in $(t_1, \ldots, t_n)$ if the variable is free with relative type σ in t_j, for some $j \in \{1, \ldots, n\}$.

The type substitution θ in Parts 4 and 5 of the definition is called the *associated* mgu.

Notation. $\mathfrak{L}$ denotes the set of all terms obtained from an alphabet and is called the *language* given by the alphabet.

Definition. A term is *closed* if it contains no free variables.

Example. Let M be a nullary type, and $A : M$ and $concatenate : List\ a \times List\ a \to List\ a$ be constants. Recall that $[] : List\ a$ and $(:) : a \to List\ a \to List\ a$ are the data constructors for lists. We will show that $(concatenate\ ([], [A]))$ is a term. For this, $([], [A])$ must be shown to be a term, which leads to the consideration of $[]$ and $[A]$. Now $[]$ is a term of type $List\ a$, by Part 2 of the definition of a term. By Parts 2 and 4, $(:\ A)$ is a term of type $List\ M \to List\ M$, where along the way the equation $a = M$ is solved with the mgu $\{a/M\}$. Then $((:\ A)\ [])$ (that is, $[A]$) is a term of type $List\ M$ by Part 4, where the equation $List\ M = List\ a$ is solved. By Part 5, it follows that $([], [A])$ is a term of type $List\ a \times List\ M$. Finally, by Part 4 again, $(concatenate\ ([], [A]))$ is a term of type $List\ M$, where the equation to be solved is $List\ a \times List\ a = List\ a \times List\ M$ whose mgu is $\{a/M\}$.

Example. Consider the constants $append : List\ a \to List\ a \to List\ a \to \Omega$ and $process : List\ a \to List\ a$. We will show that $(((append\ x)\ [])\ (process\ x))$ is a term. First, the variable x is a term of type b, where the parameter is chosen to avoid a clash in the next step. Then $(append\ x)$ is a term of type $List\ a \to List\ a \to \Omega$, for which the equation solved is $List\ a = b$. Next $((append\ x)\ [])$ is a term of type $List\ a \to \Omega$ and x has relative type $List\ a$ in $((append\ x)\ [])$. Now consider $(process\ x)$, for which the constituent parts are $process$ of type $List\ c \to List\ c$ and the variable x of type d. Thus $(process\ x)$ is a term of type $List\ c$ and x has relative type $List\ c$ in $(process\ x)$. Finally, we have to apply $((append\ x)\ [])$ to the term $(process\ x)$. For this, by Part 4, there are two equations. These are $List\ a = List\ c$, coming from the top-level types, and $List\ a = List\ c$, coming from the free variable x in each of the components. These equations have the mgu $\{c/a\}$. Thus $(((append\ x)\ [])\ (process\ x))$ is a term of type Ω.

Notation. Terms of the form $(\Sigma\ \lambda x.t)$ are written as $\exists x.t$ and terms of the form $(\Pi\ \lambda x.t)$ are written as $\forall x.t$ (in accord with the intended meaning of Σ and Π). In a higher-order logic, one may identify sets and predicates – the actual identification is between a set and its characteristic function which is a predicate. Thus, if t is of type Ω, the abstraction $\lambda x.t$ may be written as $\{x \mid t\}$ if it is intended to emphasise that its intended meaning is a set. The notation $\{\}$ means $\{x \mid \bot\}$. The notation $s \in t$ means $(t\ s)$, where t has type $\alpha \to \Omega$ and s has type α, for some α. Furthermore, notwithstanding the fact that sets are mathematically identified with predicates, it is sometimes convenient to maintain an informal distinction between sets (as "collections of objects") and predicates. For this reason, the notation $\{\alpha\}$ is introduced as a synonym for the type $\alpha \to \Omega$. The term $(s\ t)$ is often written as simply $s\ t$, using juxtaposition to denote application. Juxtaposition is left associative, so that $r\ s\ t$ means $((r\ s)\ t)$. Thus $(((\mathit{append}\ x)\ [])\ (\mathit{process}\ x))$ can be written more simply as $\mathit{append}\ x\ []\ (\mathit{process}\ x)$.

A.3 Normal Terms

Next we identify a class of terms, called *basic* terms, suitable for representing individuals in diverse applications. From a (higher-order) programming language perspective, basic terms are data values. The most interesting aspect of the class of basic terms is that it includes certain abstractions and therefore is much wider than is normally considered for knowledge representation. These abstractions allow one to model sets, multisets, and similar data types, in an elegant way. Of course, there are other ways of introducing (extensional) sets, multisets, and so on, without using abstractions. For example, one can define abstract data types or one can introduce data constructors with special equality theories. The primary advantage of the approach adopted here is that one can define these abstractions *intensionally* [Llo01b].

The definition of basic terms is given in several stages: first we define normal terms, then define an equivalence relation on normal terms, and finally define basic terms as distinguished representatives of equivalence classes. Before getting down to the first step of giving the definition of normal terms, some motivation will be helpful. How should a (finite) set or multiset be represented? First, advantage is taken of the higher-order nature of the logic to identify sets and their characteristic functions, that is, sets are viewed as predicates. With this approach, an obvious representation of sets uses the connectives, so that $\lambda x.(x = 1) \vee (x = 2)$ is the representation of the set $\{1, 2\}$. This was the kind of representation used in [Llo99] and it works well for sets. But the connectives are, of course, not available for multisets, so something more general is needed. An alternative representation for the set $\{1, 2\}$ is the term

$$\lambda x.\mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \bot$$

and this idea generalises to multisets and similar abstractions. For example,

$$\lambda x.\mathit{if}\ x = A\ \mathit{then}\ 42\ \mathit{else}\ \mathit{if}\ x = B\ \mathit{then}\ 21\ \mathit{else}\ 0$$

is the multiset with 42 occurrences of A and 21 occurrences of B (and nothing else). Thus we adopt abstractions of the form

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$$

to represent (extensional) sets, multisets, and so on.

However, before giving the definition of a normal term, some attention has to be paid to the term s_0 in previous expression. The reason is that s_0 in this abstraction is usually a very specific term. For example, for finite sets, s_0 is $\bot$ and for finite multisets, s_0 is 0. For this reason, the concept of a default term is now introduced. The intuitive idea is that, for each closed type, there is a (unique) default term such that each abstraction having that type as codomain takes the default term as its value for all but a finite number of points in the domain, that is, s_0 is the default value. The choice of default term depends on the particular application but, since sets and multisets are so useful, one would expect the set of default terms to include $\bot$ and 0. However, there could also be other types for which a default term is needed. For each type constructor T, we assume there is chosen a unique *default data constructor* C such that C has signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \dots a_k)$. For example, for Ω, the default data constructor could be $\bot$, for *Int*, the default data constructor could be 0, and for *List*, the default data constructor could be [].

Definition. The set of *default terms*, $\mathfrak{D}$, is defined inductively as follows.

1. If C is a default data constructor having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \dots a_k)$ and $t_1, \dots, t_n \in \mathfrak{D}$ $(n \geq 0)$ such that $C\ t_1 \dots t_n \in \mathfrak{L}$, then $C\ t_1 \dots t_n \in \mathfrak{D}$.
2. If $t \in \mathfrak{D}$ and $x \in \mathfrak{V}$, then $\lambda x.t \in \mathfrak{D}$.
3. If $t_1, \dots, t_n \in \mathfrak{D}$ $(n \geq 0)$ and $(t_1, \dots, t_n) \in \mathfrak{L}$, then $(t_1, \dots, t_n) \in \mathfrak{D}$.

There may not be a default term for some closed types. However, if it exists, one can show that the default term for each closed type is unique.

Now normal terms can be defined. In the following, $\lambda x.s_0$ is regarded as the special case of

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \dots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$$

when $n = 0$.

Definition. The set of *normal terms*, $\mathfrak{N}$, is defined inductively as follows.

1. If C is a data constructor having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \dots a_k)$ and $t_1, \dots, t_n \in \mathfrak{N}$ $(n \geq 0)$ such that $C\ t_1 \dots t_n \in \mathfrak{L}$, then $C\ t_1 \dots t_n \in \mathfrak{N}$.
2. If $t_1, \dots, t_n \in \mathfrak{N}$, $s_1, \dots, s_n \in \mathfrak{N}$ $(n \geq 0)$, $s_0 \in \mathfrak{D}$ and

   $$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \dots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{L},$$

   then

   $$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \dots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{N}.$$
3. If $t_1, \dots, t_n \in \mathfrak{N}$ $(n \geq 0)$ and $(t_1, \dots, t_n) \in \mathfrak{L}$, then $(t_1, \dots, t_n) \in \mathfrak{N}$.

Part 1 of the definition of the set of normal terms states, in particular, that individual natural numbers, integers, and so on, are normal terms. Also a term formed by applying a constructor to (all of) its arguments, each of which is a normal term, is a normal term. As an example of this, consider the following declarations of the data constructors *Circle* and *Rectangle*.

$$\mathit{Circle} : \mathit{Float} \to \mathit{Shape}$$
$$\mathit{Rectangle} : \mathit{Float} \to \mathit{Float} \to \mathit{Shape}.$$

Then (*Circle* 7.5) and (*Rectangle* 42.0 21.3) are normal terms of type *Shape*. However, (*Rectangle* 42.0) is not a normal term as not all arguments to *Rectangle* are given. Normal terms coming from Part 1 of the definition are called *normal structures* and always have a type of the form $T\ \alpha_1 \dots \alpha_n$.

The abstractions formed in Part 2 of the definition are "almost constant" abstractions since they take the default term s_0 as value for all except a finite number of points in the domain. They are called *normal abstractions* and always have a type of the form $\beta \to \gamma$. This class of abstractions includes useful data types such as (finite) sets and multisets (assuming $\bot$ and 0 are default terms). More generally, normal abstractions can be regarded as lookup tables, with s_0 as the value for items not in the table.

Part 3 of the definition of normal terms just states that one can form a tuple from normal terms and obtain a normal term. These terms are called *normal tuples* and always have a type of the form $\alpha_1 \times \cdots \times \alpha_n$.

Proposition A.1. $\mathfrak{D} \subseteq \mathfrak{N}$.

It will be convenient to gather together all normal terms that have a type more general than some specific closed type.

Definition. For each $\alpha \in \mathfrak{S}^c$, define $\mathfrak{N}_\alpha = \{t \in \mathfrak{N} \mid t \text{ has type more general than } \alpha\}$.

The intuitive meaning of $\mathfrak{N}_\alpha$ is that it is the set of terms representing individuals of type α. Note that $\mathfrak{N} = \bigcup_{\alpha \in \mathfrak{S}^c} \mathfrak{N}_\alpha$. However, the $\mathfrak{N}_\alpha$ are not necessarily disjoint. For example, if the alphabet includes *List*, then $[] \in \mathfrak{N}_{List\ \alpha}$, for each closed type α.

A.4 An Equivalence Relation on Normal Terms

Several syntactically distinct terms in $\mathfrak{N}$ can represent the same individual. For example,

$$\lambda x.\mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \bot,$$
$$\lambda x.\mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \bot,$$

and

$$\lambda x.\mathit{if}\ x = 3\ \mathit{then}\ \bot\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \bot$$

all represent the set $\{1, 2\}$. To reflect this, a relation $\equiv$ is defined on $\mathfrak{N}$.

Definition. The binary relation $\equiv$ on $\mathfrak{N}$ is defined inductively as follows. Let $s, t \in \mathfrak{N}$. Then $s \equiv t$ if there exists $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{N}_\alpha$ and one of the following conditions holds.

1. $\alpha = T\ \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$, and s is $C\ s_1 \ldots s_n$, t is $C\ t_1 \ldots t_n$ and $s_i \equiv t_i$, for $i = 1, \ldots, n$.

2. $\alpha = \beta \to \gamma$, for some β, γ, and s is $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$,
   t is $\lambda y.\mathit{if}\ y = u_1\ \mathit{then}\ v_1\ \mathit{else}\ \ldots\ \mathit{if}\ y = u_m\ \mathit{then}\ v_m\ \mathit{else}\ s_0$
   and, $\forall r \in \mathfrak{N}_\beta$,
   $(\exists i, j.\ r \equiv t_i\ \wedge\ r \not\equiv t_k (\forall k < i)\ \wedge\ r \equiv u_j\ \wedge\ r \not\equiv u_m (\forall m < j)\ \wedge\ s_i \equiv v_j) \vee$
   $(\exists i.\ r \equiv t_i\ \wedge\ r \not\equiv t_k (\forall k < i)\ \wedge\ r \not\equiv u_j (\forall j)\ \wedge\ s_i \equiv s_0) \vee$
   $(\exists j.\ r \not\equiv t_i (\forall i)\ \wedge\ r \equiv u_j\ \wedge\ r \not\equiv u_m (\forall m < j)\ \wedge\ s_0 \equiv v_j) \vee$
   $(r \not\equiv t_i (\forall i)\ \wedge\ r \not\equiv u_j (\forall j))$.

3. $\alpha = \alpha_1 \times \cdots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$, and s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and $s_i \equiv t_i$, for $i = 1, \ldots, n$.

Proposition A.2. *For each* $\alpha \in \mathfrak{S}^c$, $\equiv|_{\mathfrak{N}_\alpha}$ *is an equivalence relation on* $\mathfrak{N}_\alpha$.

A.5 A Total Order on Normal Terms

The equivalence relation $\equiv$ was introduced because several syntactically distinct terms in $\mathfrak{N}$ can represent the same individual. Rather than deal with all the normal terms in an equivalence class in some $\mathfrak{N}_\alpha$, it is preferable to deal with a single representative from the equivalence class. For this purpose, a (strict) total order on normal terms is introduced.

Recall that a (strict) partial order on a set A is a binary relation $<$ on A such that, for each $a, b, c \in A$, $a \not< a$ (irreflexivity), $a < b$ implies $b \not< a$ (asymmetry), and $a < b$ and $b < c$ implies $a < c$ (transitivity). In addition, a (strict) partial order is a (strict) total order if, for each $a, b \in A$, exactly one of $a = b$ or $a < b$ or $b < a$ holds.

If $<$ is a (strict) total order on a set A, then $<$ can be lifted to (strict) total order, also denoted by $<$, on the set of sequences of elements in A by $a_1 \ldots a_n < b_1 \ldots b_m$ if either
(i) $a_1 = b_1, \ldots, a_n = b_n$ and $n < m$, or

(ii) there exists j such that $1 \leq j \leq n$, $a_1 = b_1, \ldots, a_{j-1} = b_{j-1}$ and $a_j < b_j$.
The order $<$ on the sequences is called the *induced lexicographic* ordering.

In the definition of the binary relation $<$ below, it is assumed that, for each $T \in \mathfrak{T}$, there is defined a (strict) total order $\prec_T$ on the set of all data constructors associated with the type constructor T. For standard types, such as *Int* and *Float*, the usual order provides an appropriate total order. To simplify the statement of the definition, the concept of the trace of an abstraction will be useful.

Definition. Suppose that s is a normal abstraction

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0.$$

Then the *trace* of s, *trace*(s), is the sequence $t_1\, s_1\, t_2\, s_2 \ldots t_n\, s_n$. (For the normal abstraction $\lambda x.s_0$, the trace is the empty sequence ε.)

Definition. The binary relation $<$ on $\mathfrak{N}$ is defined inductively as follows. Let $s, t \in \mathfrak{N}$. Then $s < t$ if there exists $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{N}_\alpha$ and one of the following conditions holds.

1. $\alpha = T\ \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$, and s is $C\ s_1 \ldots s_n$, t is $D\ t_1 \ldots t_m$ and either $C \prec_T D$ or $C = D$ and there exists j such that $1 \leq j \leq n$, $s_1 = t_1, \ldots, s_{j-1} = t_{j-1}$ and $s_j < t_j$.

2. $\alpha = \beta \to \gamma$, for some β, γ, and *trace*$(s) <$ *trace*(t), where $<$ is the induced lexicographic ordering.

3. $\alpha = \alpha_1 \times \cdots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$, and s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and there exists j such that $1 \leq j \leq n$, $s_1 = t_1, \ldots, s_{j-1} = t_{j-1}$ and $s_j < t_j$.

Proposition A.3. *For each $\alpha \in \mathfrak{S}^c$, $<|_{\mathfrak{N}_\alpha}$ is a (strict) total order on $\mathfrak{N}_\alpha$.*

A.6 Basic Terms

Finally, the definition of the key concept of a basic term can be given.

Definition. The set of *basic terms*, $\mathfrak{B}$, is defined inductively as follows.

1. If C is a data constructor having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$ and $t_1, \ldots, t_n \in \mathfrak{B}$ $(n \geq 0)$ such that $C\ t_1 \ldots t_n \in \mathfrak{L}$, then $C\ t_1 \ldots t_n \in \mathfrak{B}$.

2. If $t_1, \ldots, t_n \in \mathfrak{B}$, $s_1, \ldots, s_n \in \mathfrak{B}$, $t_1 < \ldots < t_n$, $s_i \notin \mathfrak{D}$, for $1 \leq i \leq n$ $(n \geq 0)$, $s_0 \in \mathfrak{D}$ and

   $$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{L},$$

   then

   $$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{B}.$$

3. If $t_1, \ldots, t_n \in \mathfrak{B}$ $(n \geq 0)$ and $(t_1, \ldots, t_n) \in \mathfrak{L}$, then $(t_1, \ldots, t_n) \in \mathfrak{B}$.

The basic terms from Part 1 of the definition are called *basic structures*, those from Part 2 are called *basic abstractions*, and those from Part 3 are called *basic tuples*.

Proposition A.4. $\mathfrak{D} \subseteq \mathfrak{B} \subseteq \mathfrak{N}$.

As for normal terms, the basic terms of a particular type can be gathered together.

Definition. For each $\alpha \in \mathfrak{S}^c$, define $\mathfrak{B}_\alpha = \{t \in \mathfrak{B} \mid t \text{ has type more general than } \alpha\}$.

Proposition A.5. *For each $\alpha \in \mathfrak{S}^c$, $\mathfrak{B}_\alpha \subseteq \mathfrak{N}_\alpha$.*

The next result shows that, for basic terms, the equivalence relation $\equiv$ reduces to the identity relation.

Proposition A.6. *Let $s, t \in \mathfrak{B}$. Then $s \equiv t$ iff $s = t$.*

The next proposition justifies restricting attention to basic terms for knowledge representation purposes.

Proposition A.7. *If $s \in \mathfrak{N}_\alpha$, for some $\alpha \in \mathfrak{S}^c$, then there is a unique $t \in \mathfrak{B}_\alpha$ such that $s \equiv t$.*

Definition. Let $s \in \mathfrak{N}_\alpha$, for some $\alpha \in \mathfrak{S}^c$. The unique $t \in \mathfrak{B}_\alpha$ such that $s \equiv t$ is called the *basic form* of s.

Here is an example to illustrate how the basic form can be computed.

Example. Let s be the normal term

$$\lambda x.\mathit{if}\ x = 3\ \mathit{then}\ \bot\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 3\ \mathit{then}\ \top\ \mathit{else}\ \bot.$$

Assume that the total order on the integers is the usual order. In the first step, each of $1, 2, 3$, and $\top$ and $\bot$ are replaced by their basic form. Since each of these is already a basic term, this step has no effect. Second, the component of the if-then-else containing the duplicated occurrence of $x = 3$ is dropped to obtain

$$\lambda x.\mathit{if}\ x = 3\ \mathit{then}\ \bot\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \bot.$$

Third, the component containing the occurrence $x = 3$ is dropped since the corresponding value is $\bot$ to obtain

$$\lambda x.\mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \bot.$$

Finally, the sequence 2 1 is ordered according to the total order and the components of the if-then-else are reordered accordingly to obtain

$$\lambda x.\mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \bot,$$

which is the basic form of s.

A.7 Transformations

Having established the use of the logic for representing individuals, we now turn to the problem of constructing predicates that individuals may or may not satisfy. Essentially, all that is required is the definition of a suitable collection of predicates on the type of an individual. However, these predicates are usually built up incrementally by composition and it is this incremental construction that we study here.

Composition is handled by the (reverse) composition function

$$\cdot : (\rho \to \sigma) \to (\sigma \to \tau) \to (\rho \to \tau)$$

defined by

$$((f \cdot g)\ x) = (g\ (f\ x)).$$

Predicates are built up by composing transformations, which are defined as follows.

Definition. A *transformation* f is a function having a signature of the form

$$f : (\rho_1 \to \Omega) \to \cdots \to (\rho_k \to \Omega) \to \mu \to \sigma,$$

where any parameters in $\rho_1, \ldots, \rho_k$ and σ appear in μ, and $k \geq 0$. The type μ is distinguished and is called the *source* of the transformation, while the type σ is called the *target* of the transformation. The number k is called the *rank* of the transformation.

Here are a couple of examples of transformations in addition to those introduced in Section 4.

Example. The transformation

$$\sim : (\mu \to \Omega) \to \mu \to \Omega$$

defined by

$$\sim\ p = \lambda x.\neg(p\ x),$$

provides negation. But negation can also be introduced directly by the transformation

$$\neg : \Omega \to \Omega$$

which is the usual negation connective. Then the transformation $(\sim\ p)$ can be written equivalently as $p \cdot \neg$. (In practice, just one of these methods of introducing negation would be used.)

Example. Consider again the type *Shape* and the data constructors

$$\mathit{Circle} : \mathit{Float} \to \mathit{Shape}$$
$$\mathit{Rectangle} : \mathit{Float} \to \mathit{Float} \to \mathit{Shape}.$$

The function $\mathit{isCircle} : (\mathit{Float} \to \Omega) \to \mathit{Shape} \to \Omega$ defined by

$$\mathit{isCircle}\ b\ t = \exists x.(t = \mathit{Circle}\ x) \wedge (b\ x)$$

is a transformation. Similarly, the function $\mathit{isRectangle} : (\mathit{Float} \to \Omega) \to (\mathit{Float} \to \Omega) \to \mathit{Shape} \to \Omega$ defined by

$$\mathit{isRectangle}\ b\ c\ t = \exists x.\exists y.(t = \mathit{Rectangle}\ x\ y) \wedge (b\ x) \wedge (c\ y)$$

is a transformation. For predicates b and c, $(\mathit{isRectangle}\ b\ c)$ is a predicate on geometrical shapes which returns $\top$ iff the shape is a rectangle whose length satisfies b and whose breadth satisfies c.

A.8 Standard Predicates

Next the definition of the class of predicates formed by composing transformations is presented. First, a larger class of predicates, the standard predicates, is defined and then the desired subclass, the regular predicates, which has much less redundancy, is defined. In the following definition, it is assumed that some (possibly infinite) class of transformations is given and all transformations considered are taken from this class. A standard predicate is defined by induction on the number of transformations it contains as follows.

Definition. A *standard predicate* is a term of the form

$$(f_1\ b_{1,1} \ldots b_{1,k_1}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n}),$$

where f_i is a transformation of rank k_i $(i = 1, \ldots, n)$, the target of f_n is Ω, b_{i,j_i} is a standard predicate $(i = 1, \ldots, n,\ \ j_i = 1, \ldots, k_i)$, $k_i \geq 0$ $(i = 1, \ldots, n)$ and $n \geq 1$.

The set of all standard predicates is denoted by $\mathbf{S}$.

Definition. For each $\alpha \in \mathfrak{S}^c$, define $\mathbf{S}_\alpha = \{p \in \mathbf{S} \mid p$ has type $\mu \to \Omega$ and μ is more general than $\alpha\}$.

The intuitive meaning of $\mathbf{S}_\alpha$ is that it is the set of all predicates of a particular form on individuals of type α.

The implication preorder $\Leftarrow$ on $\mathbf{S}$ is defined as follows. Let $\mathcal{B}$ denote the background theory which consists of the definitions of the transformations (and associated functions).

Definition. Let $p, q \in \mathbf{S}$. Then $p \Leftarrow q$ if the type of q is more general than the type of p and $\forall x.((p\ x) \longleftarrow (q\ x))$ is a logical consequence of $\mathcal{B}$.

Proposition A.8. *For each $\alpha \in \mathfrak{S}^c$, $\Leftarrow|_{\mathbf{S}_\alpha}$ is a preorder on $\mathbf{S}_\alpha$.*

The class of standard predicates just defined contains some redundancy in that there are syntactically distinct predicates that are semantically equivalent, in the sense of the following definition.

Definition. Let $p, q \in \mathbf{S}$. Then p and q are *semantically equivalent* if the types of p and q are the same (up to variants) and $p = q$ is a logical consequence of $\mathcal{B}$.

Semantic equivalence is an equivalence relation.

Example. Without spelling out the background theory $\mathcal{B}$, one would expect that the standard predicates $(\wedge_2\ p\ q)$ and $(\wedge_2\ q\ p)$ are semantically equivalent, where p and q are standard predicates. Similarly, $(\mathit{setExists}_2\ p\ q)$ and $(\mathit{setExists}_2\ q\ p)$ are semantically equivalent. Less obviously, $(\mathit{domCard}\ p) \cdot (> 0)$ and $(\mathit{setExists}_1\ p)$ are semantically equivalent.

It is important to remove as much of this redundancy in standard predicates as possible. Ideally, one would like to be able to determine just *one* representative from each class of equivalent predicates. However, determining the semantically equivalent predicates is undecidable, so one usually settles for some easily checked syntactic conditions that reveal equivalence of predicates. Thus these syntactic conditions are sufficient, but not necessary, for equivalence. These considerations motivate the next definition.

Definition. A transformation f is *symmetric* if it has a signature of the form

$$f : (\rho \to \Omega) \to \cdots \to (\rho \to \Omega) \to \mu \to \sigma,$$

and $f\ p_1 \ldots p_k$ and $f\ p_{i_1} \ldots p_{i_k}$ are semantically equivalent, for all standard predicates $p_j : \rho \to \Omega$, for $j = 1, \ldots, k$ and permutations i of $\{1, \ldots, k\}$, where k is the rank of f.

Clearly, every transformation of rank k, where $k \leq 1$, is (trivially) symmetric. Furthermore, the transformations $\mathit{setExists}_n$ and $\wedge_n$ are symmetric. However, *isRectangle* is not symmetric since it distinguishes the length argument from the breadth argument.

Since any permutation of the predicate arguments of a symmetric transformation produces a semantically equivalent predicate, it is advisable to choose one particular order of arguments and ignore the others. For this purpose, a total order on standard predicates is defined and then arguments for symmetric transformations are chosen in increasing order according to this total order. To define the total order on standard predicates, one must start with a total order on transformations. Therefore, it is supposed that, for each type μ, the transformations having source μ are ordered according to some (arbitrary) strict total order $<$.

The following definition of the relation $p \prec q$ uses induction on the maximum of the number of transformations in p and the number in q.

Definition. The binary relation $\prec$ on $\mathbf{S}$ is defined inductively as follows. Let $p, q \in \mathbf{S}$, where p is $(f_1\ b_{1,1} \ldots b_{1,k_1}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n})$ and q is $(g_1\ c_{1,1} \ldots c_{1,s_1}) \cdot \ldots \cdot (g_r\ c_{r,1} \ldots c_{r,s_r})$. Then $p \prec q$ if there exists $\alpha \in \mathfrak{S}^c$ such that $p, q \in \mathbf{S}_\alpha$ and one of the following holds.

1. p is a (strict) prefix of q.
2. There exists i such that $f_i < g_i$, and p and q agree to the left of f_i and g_i.
3. There exist i and j_i such that $b_{i,j_i} \prec c_{i,j_i}$, and p and q agree to the left of b_{i,j_i} and c_{i,j_i}.

Proposition A.9. *For each $\alpha \in \mathfrak{S}^c$, $\prec|_{\mathbf{S}_\alpha}$ is a (strict) total order on $\mathbf{S}_\alpha$.*

The relation $\preceq$ is defined by $p \preceq q$ if either $p = q$ or $p \prec q$. Clearly, $\preceq$ is a total order on each $\mathbf{S}_\alpha$.

A.9 Regular Predicates

Now the class of regular predicates can be defined by induction on the number of transformations in a predicate.

Definition. A standard predicate $(f_1\ b_{1,1} \ldots b_{1,k_1}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n})$ is *regular* if b_{i,j_i} is a regular predicate, for $i = 1, \ldots, n$ and $j_i = 1, \ldots, k_i$, and f_i is symmetric implies that $b_{i,1} \preceq \cdots \preceq b_{i,k_i}$, for $i = 1, \ldots, n$.

The set of all regular predicates is denoted by $\mathbf{R}$.

Example. The standard predicate

$$\mathit{vertices} \cdot (\mathit{domCard}\ (\mathit{vertex} \cdot (\wedge_2\ b\ c))) \cdot (> 7)$$

is a regular predicate iff $b \preceq c$, and b and c are regular predicates.

The next result shows that each standard predicate is semantically equivalent to a regular predicate and hence attention can be confined to the generally much smaller class of regular predicates.

Proposition A.10. *For every $p \in \mathbf{S}$, there exists $q \in \mathbf{R}$ such that p and q are semantically equivalent.*

One cannot expect the regular predicate q in Proposition A.10 to be (syntactically) unique. To see this, simply note that $(\mathit{domCard}\ \mathit{top}) \cdot (> 0)$ and $(\mathit{setExists}_1\ \mathit{top})$ are regular predicates, and are semantically equivalent.

On the other hand, using the proof of Proposition A.10 as a basis, one can give an algorithm for constructing a regular predicate semantically equivalent to some given standard predicate. This algorithm is given in Figure 13 below. The regular predicate q given by this algorithm is called the *regularisation* of the standard predicate p. The regularisation of p is denoted by $\overline{p}$. The proof of Proposition A.10 shows that the regularisation $\overline{p}$ of a standard predicate p is in fact a regular predicate, and $\overline{p}$ and p are semantically equivalent. It is also clear that the regularisation of a regular predicate is itself.

function *Regularise*(p) **returns** a regular predicate q such that p and q are semantically equivalent;

input: p, a standard predicate $(f_1\ b_{1,1} \ldots b_{1,k_1}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n})$;

for $i = 1, \ldots, n$ **do**
    for $j_i = 1, \ldots, k_i$ **do**
        $\overline{b_{i,j_i}} := \mathit{Regularise}(b_{i,j_i})$;
    if f_i is symmetric
    then
        $[c_{i,1}, \ldots, c_{i,k_i}] := \mathit{Sort}([\overline{b_{i,1}}, \ldots, \overline{b_{i,k_i}}])$;
        $p_i := (f_i\ c_{i,1} \ldots c_{i,k_i})$;
    else
        $p_i := (f_i\ \overline{b_{i,1}} \ldots \overline{b_{i,k_i}})$;
return $p_1 \cdot \ldots \cdot p_n$;

Figure 13: Algorithm for regularising a standard predicate