

A Tutorial Introduction to ALKEMY

K.S. Ng

Computer Sciences Laboratory
The Australian National University

February 23, 2006

1 Introduction

This paper serves as a tutorial introduction to ALKEMY, a decision-tree learner designed to induce comprehensible theories from structured data. A short and clear account of the motivations behind the development of the system is in [FGCL98]. [BGCL01] gives a more detailed discussion of the relevant knowledge representation issues and presents the main algorithms. A rigorous treatment of the logic underlying the system is in [Llo03]. Aspects of the theory of learning with Alkemy can be found in [Ng05]. I refer the reader to these articles for more information on ALKEMY. In this paper, we will concentrate on the practice of learning how to use a prototype implementation.

The tutorial is organised as follows. In §2, we introduce in simple terms the central concepts behind the learner. We review the standard decision-tree learning algorithm and point out the essential differences between ALKEMY and more conventional learners like CART [BFOS84] and C4.5 [Qui93]. In §3, we examine in some detail a recent implementation of ALKEMY. Among other things, we look at the input and output formats of the system, and explore some of its functionalities. To make the ideas concrete, we use a classic problem in symbolic learning as a running example throughout §2 and §3.

2 Alkemy: The Concept

The basic problem we are trying to solve is standard. The learner receives randomly drawn training examples of the form (x, y) , $x \in \mathcal{X}$, $y \in \mathcal{Y}$, where \mathcal{X} is some set called the *individual* space, and \mathcal{Y} is either a finite set or (a bounded interval of) the real line \mathbb{R} . There could be an underlying target function f relating each x and y by $y = f(x)$, and the examples are generated according to an unknown probability distribution on \mathcal{X} . More generally, we do not presuppose the existence of a target function and simply assume that examples are generated independently according to an unknown joint probability distribution on $\mathcal{X} \times \mathcal{Y}$. In both cases, the learning task is to find a hypothesis h from a class of functions \mathcal{H} called the hypothesis space that generalizes well to unseen examples.

To learn a decision tree from a set of labelled examples $\mathcal{E} = \{(x_i, y_i)\}_{i=1}^n$, a set of (useful) questions \mathcal{Q} one can ask about the individuals (the x_i 's) needs to be identified. Each question is a function mapping \mathcal{X} to some finite domain. The learning algorithm then proceeds as follows. Starting from the single-node tree T with the whole training set \mathcal{E} in it, a search is performed to find a question $q \in \mathcal{Q}$ that best partitions, according to some pre-specified measure of success, the initial set \mathcal{E} into subsets $\mathcal{E}_1, \dots, \mathcal{E}_k$. Here, the question q has k possible outcomes, and the partition $\mathcal{P} = (\mathcal{E}_1, \dots, \mathcal{E}_k)$ is in some sense *purser* than \mathcal{E} . We then put q as the discriminating function in T and attach k nodes to it, assigning \mathcal{E}_i to the i -th node. The process is then repeated on each of the newly-created leaf nodes and continued on recursively on its descendants. The process terminates when the subset of examples falling in each leaf node in the tree has attained a

certain minimum level of purity. Each leaf node is then labelled appropriately in accordance with the examples falling in it.

ALKEMY is different from conventional decision-tree learners in two important aspects — the language that can be used to represent the x_i 's, and the kind of questions \mathcal{Q} one can pose about individuals so-represented. For learners in the C4.5/CART family, the x_i 's are feature vectors with categorical and/or continuous attributes. In the case of ALKEMY, they are a special class of terms called basic terms in a typed higher-order logic. The use of such a rich language allows us to upgrade, in the sense expounded in [VD01], standard top-down decision-tree learning algorithms to handle structured data. For example, ALKEMY can accept as input just about every commonly encountered data type in computer science. These include booleans, characters, string, integers, floating-point numbers, lists, trees, sets, multisets, graphs, and composite types that can be built up from these.

The fact that we are proposing to use a highly expressive language to represent the individuals means that the kind of questions \mathcal{Q} we can ask about them has to be correspondingly rich as well. A class of predicates (boolean-valued functions) in the logic is identified for that purpose. Associated with each simple type are some basic operations. For example, given a tuple, we can project onto each of its components; given a set, we can compute its cardinality, among other things; given a graph, we can pull out the set of all connected subgraphs with five vertices in it; etc. The basic idea is that we can form complex predicates on individuals by composing these simple functions. We will make concrete the ideas introduced so far with an example.

Consider the East-West challenge proposed by Michalski, as illustrated in Figure 1. We are given 10 trains and the directions they are travelling in. The problem is to produce a rule that can differentiate between those heading east and those heading west.

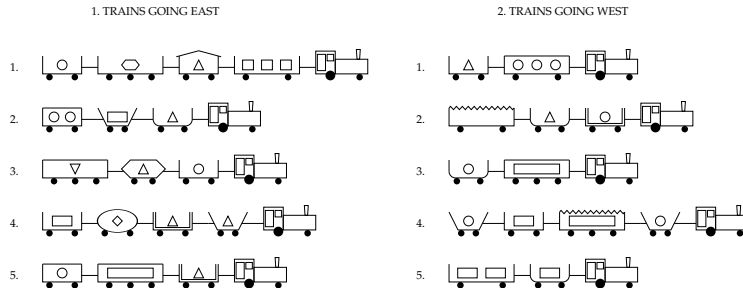


Figure 1: Trains in Michalski's East-West Challenge.

To solve the problem using ALKEMY, one must first decide how the trains are to be represented. Here, we model a train as a list of its carriages. A carriage is in turn modelled using a tuple made up using six of its properties: shape, length, number of wheels, open air or closed, roof shape, and the load the carriage is carrying. From that initial analysis, we introduce the following data constructors

- Rectangular, DoubleRectangular, UShaped, BucketShaped, Hexagonal, Ellipsoidal* : Shape
- Long, Short* : Length
- Closed, Open* : Kind
- Flat, Jagged, Peaked, Curved, None* : Roof
- Circle, Hexagon, Square, Rectangle, LRectangle, Triangle, UTriangle, Diamond, Null* : Object
- East, West* : Direction

and the following type synonyms

$$\begin{aligned} \text{NumWheels} &= \text{Int} \\ \text{NumObjects} &= \text{Int} \\ \text{Load} &= \text{Object} \times \text{NumObjects} \\ \text{Car} &= \text{Shape} \times \text{Length} \times \text{NumWheels} \times \text{Kind} \times \text{Roof} \times \text{Load} \\ \text{Train} &= \text{List Car}. \end{aligned}$$

A note on notation. In the declaration of the data constructors, the string appearing after the colon denotes the *type* of the constants appearing before it. Type synonyms allows us to give meaningful names to types.

We want to learn a function *direction* having the following signature.

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

The first trains travelling in each direction are shown below.

$$\begin{aligned} \text{direction} &[(\text{Rectangular}, \text{Long}, 2, \text{Open}, \text{None}, (\text{Square}, 3)), \\ &(\text{Rectangular}, \text{Short}, 2, \text{Closed}, \text{Peaked}, (\text{Triangle}, 1)), \\ &(\text{Rectangular}, \text{Long}, 3, \text{Open}, \text{None}, (\text{Hexagon}, 1)), \\ &(\text{Rectangular}, \text{Short}, 2, \text{Open}, \text{None}, (\text{Circle}, 1))] = \text{East} \\ \text{direction} &[(\text{Rectangular}, \text{Long}, 2, \text{Closed}, \text{Flat}, (\text{Circle}, 3)), \\ &(\text{Rectangular}, \text{Short}, 2, \text{Open}, \text{None}, (\text{Triangle}, 1))] = \text{West}. \end{aligned}$$

Having decided on the representation, the next step is to identify the set \mathcal{Q} , *i.e.*, the hypothesis language. We first give the definition of a transformation. The class of transformations identified for a particular application forms the basic class of functions with which we compose to form more complex predicates on individuals.

Definition 2.1. A *transformation* f is a function having a signature of the form

$$f : (\rho_1 \rightarrow \Omega) \rightarrow \dots \rightarrow (\rho_k \rightarrow \Omega) \rightarrow \mu \rightarrow \sigma,$$

where any parameters (*i.e.*, type variables) in ρ_1, \dots, ρ_k and σ appear in μ , and $k \geq 0$. The type μ 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.

What transformations are suitable for the learning task? We note here that the type we have chosen to represent the trains gives strong clues about the possible transformations. At the top-level, a train is a list of carriages. Some useful transformations on lists are:

$$\begin{aligned} \text{head} &: \text{Train} \rightarrow \text{Car} \\ \text{head} (\# x y) &= x \\ \text{tail} &: \text{Train} \rightarrow \text{Train} \\ \text{tail} (\# x y) &= y \\ \text{listToSet} &: \text{Train} \rightarrow \{\text{Car}\} \\ \text{listToSet} [] &= \{\} \\ \text{listToSet} (\# x y) &= \{x\} \cup (\text{listToSet } y) \end{aligned}$$

Here, we use a functional logic programming language called Escher [Llo95, Llo99] to define the functions. Escher can be most straightforwardly understood as a simple extension of Haskell [Tho99] to incorporate logic programming. Appendix A contains a short introduction to Escher.

The function *head* returns the head of a list; the function *tail* returns the tail of a list; and finally, the function *listToSet* takes as input a list and converts it into a set.

Given a set of carriages, we find the following transformation helpful.

$$\begin{aligned} \text{setExists}_1 &: (Car \rightarrow \Omega) \rightarrow \{Car\} \rightarrow \Omega \\ \text{setExists}_1 p t &= \exists x.(x \in t \wedge (p x)) \end{aligned}$$

The transformation returns true given a set iff the set has at least one element that satisfies the predicate *p*. Here and in the following, the symbol Ω denotes the type of the booleans. Sets are identified with predicates in the logic. Thus, both *p* and *t* are predicates on *Car*. In that sense, there is no mathematical distinction between $Car \rightarrow \Omega$ and $\{Car\}$. The notation is introduced here to emphasise the intuitive roles the corresponding terms are playing. The former is a condition, the latter a collection of objects. So we could have written $(t x)$ in place of $x \in t$ in the above and the meaning of the function would stay unchanged. For a more detailed discussion on this point, the reader is referred to page 25 and page 136 of [Llo03].

Corresponding to the types *Car* and *Load*, which are tuples, are projection transformations.

$$\begin{aligned} \text{projShape} &: Car \rightarrow Shape \\ \text{projShape} (t_0, t_1, t_2, t_3, t_4, t_5) &= t_0 \\ \dots & \\ \text{projLoad} &: Car \rightarrow Load \\ \text{projLoad} (t_0, t_1, t_2, t_3, t_4, t_5) &= t_5 \\ \\ \text{projObject} &: Load \rightarrow Object \\ \text{projObject} (t_0, t_1) &= t_0 \\ \\ \text{projNumObjects} &: Load \rightarrow NumObjects \\ \text{projNumObjects} (t_0, t_1) &= t_1 \end{aligned}$$

We may also be interested in testing the conjunctions of several predicates defined on *Car*.

$$\begin{aligned} \wedge_2 &: (Car \rightarrow \Omega) \rightarrow (Car \rightarrow \Omega) \rightarrow Car \rightarrow \Omega \\ \wedge_2 p_1 p_2 &= \lambda x.((p_1 x) \wedge (p_2 x)) \end{aligned}$$

Finally, corresponding to the data constructors and integers, we have the following transformations that check for equality of terms.

$$\begin{aligned} (= \text{Rectangular}) &: Shape \rightarrow \Omega \\ (= \text{Rectangular}) x &= (x = \text{Rectangular}) \\ \dots & \\ (= \text{Null}) &: Object \rightarrow \Omega \\ (= \text{Null}) x &= (x = \text{Null}) \\ \\ (= 2) &: NumWheels \rightarrow \Omega \\ (= 2) x &= (x = 2) \\ \\ (= 3) &: NumWheels \rightarrow \Omega \\ (= 3) x &= (x = 3) \end{aligned}$$

In preparation for discussions on predicate rewrite systems, we also define here the transformation $\text{top} : \mu \rightarrow \Omega$ that always return true, that is, $\text{top } x = \top$.

As mentioned earlier, predicates on individuals are built up through compositions of transformations. Composition is handled by the (reverse) composition function.

$$\begin{aligned} \circ & : (\rho \rightarrow \sigma) \rightarrow (\sigma \rightarrow \tau) \rightarrow (\rho \rightarrow \tau) \\ (f \circ g) x & = (g (f x)). \end{aligned}$$

We next state the class of predicates we want to generate, the standard predicates. It is defined by induction on the number of transformations taken from a class identified for a particular application under consideration.

Definition 2.2. A *standard predicate* is a term of the form

$$(f_1 p_{1,1} \dots p_{1,k_1}) \circ \dots \circ (f_n p_{n,1} \dots p_{n,k_n}),$$

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

As an example, $listToSet \circ (setExists_1 (projLength \circ (= Short)))$ is a standard predicate that takes as input an individual of type *Train*, converts it into a set of carriages, and returns true iff there exists a short carriage in the set.

Having identified the class of predicates we want to construct, the next question of interest is how do we (compactly) define and (efficiently) enumerate the predicates? A grammar-like construct called a predicate rewrite system is employed for this purpose. This is introduced next.

To define a space of predicates, we first need to make some conjectures about the form of the hypothesis. Suppose we speculate that the definition of *direction* has the following general form: a train travels east (or west) iff there exists a carriage in the train that satisfies some (as yet unknown) properties. Based on that, we can define the following predicate rewrite system.

$$\begin{aligned} top & \mapsto listToSet \circ (setExists_1 top) \\ top & \mapsto \wedge_2 top top \\ top & \mapsto projShape \circ top \\ top & \mapsto projLength \circ top \\ top & \mapsto projNumWheels \circ top \\ top & \mapsto projKind \circ top \\ top & \mapsto projRoof \circ top \\ top & \mapsto projLoad \circ top \\ top & \mapsto projObjects \circ top \\ top & \mapsto projNumObjects \circ top \\ top & \mapsto (= Rectangular) \\ & \dots \\ top & \mapsto (= Null) \end{aligned}$$

Each $s \mapsto t$ can be most straightforwardly understood as a rewrite rule. Given a predicate on the LHS, we can replace it with the RHS provided certain conditions hold. For example, given a predicate $p = q \circ r \circ s$, we can use the rule $s \mapsto t$ to rewrite p , denoted $p[s/t]$, to produce $p' = q \circ r \circ t$, provided the resulting p' is a term in the logic. In that sense, the set of rewrites defines a structured search space as shown in Figure 2.

Rewrite systems are usually built based on equality. In the case of ALKEMY, the basic concept is not equality, but implication. In simplistic terms, given a rewrite $r \mapsto s$ where s logically implies r (denoted $s \longrightarrow r$) and a predicate p where r is a subterm in p , when we use $r \mapsto s$ on p to obtain a new predicate $p[r/s]$, it is always the case that $p[r/s] \longrightarrow p$. Looking back at Figure 2, we note that if there is an arc from a predicate p to a predicate q , then it follows that $q \longrightarrow p$. This property allows us to do general-to-specific hypothesis searches. This property, combined

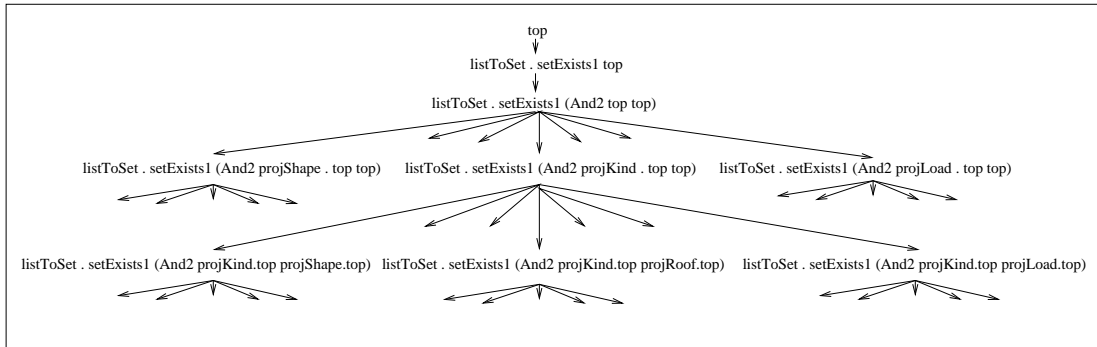


Figure 2: The search space illustrated.

with predicate selection heuristics described in [NL05], is also important for effective pruning of search spaces during learning.

Given the representation, transformations and predicate rewrite system above, ALKEMY found the following definition for the function *direction*.

```

direction t =
  if listToSet ◦ (setExists1 (∧2 (projLength ◦ (= Short)) (projKind ◦ (= Closed)))) t
  then East
  else West.

```

In a more ‘brain-friendly’ language: “A train is eastbound if and only if it has a short closed car”.

Bibliographical Notes Most early pioneering work on decision-tree learning are well covered in the classic textbooks [BFOS84] and [Qui93]. Two systematic generalisations of the algorithm to the first-order relational setting, work similar in spirit to ALKEMY, are described in [Blo98, BD98] and [KW01, Kra96]. The subfield of machine learning most relevant to the present work is Inductive Logic Programming (ILP), an area of research concerned with learning comprehensible rules. If ILP is defined as the intersection of inductive learning and logic programming [MD94], then our work can be thought of as one of the first children in the marriage between inductive learning and functional logic programming [Llo95, Llo99].

3 Alkemy: An Implementation

3.1 Obtaining Alkemy

ALKEMY is available for download from <http://rsise.anu.edu.au/~jwl/LogicforLearning>. The file README.1ST contains installation instructions and (potential) solutions to some anticipated problems. A complete listing of ALKEMY’s source code in the form of a literate program [Ng06] comes equipped with the system. It is the file `alkemy.ps.gz` in the top-level directory.

The ALKEMY system can be accessed in the usual way from the command line. Thanks to Matt Gray, we also have a graphical user interface. It is implemented in TCL/TK, and can be found in the `gui` directory. Versions for MacOS X and XWindows are available. For some, this may be the preferred mode of operation.

3.2 A Whirlwind Tour

This section gives a quick tour of the user interface, using the now familiar East-West challenge as an example. We first describe the format of the input specification file, followed by the format

and meaning of system outputs.

3.2.1 The Input Specification File

In ALKEMY's language, this is what the East-West challenge classification task looks like. The now familiar Escher syntax is employed for declaring data constructors and defining functions.

```
%% -- Data declarations

Circle, Hexagon, Square, Rectangle, LongRectangle, Triangle, UTriangle,
                                                Diamond, Null : Object ;

Flat, Jagged, Peaked, Curved, None : Roof ;
Rectangular, DoubleRectangular, UShaped, BucketShaped, Hexagonal,
                                                Ellipsoidal : Shape ;

Long, Short : Length ;
Closed, Open : Kind ;
East, West : Class ;
type NumWheels = Int ;
type NumObjects = Int ;
type Load = (Object * NumObjects) ;
type Car = (Shape * Length * NumWheels * Kind * Roof * Load) ;
type Individual = (List Car) ;

LEARN direction : Individual -> Class ;

%% -- Training examples

direction [ (Rectangular, Long, 2, Open, None, (Square, 3)),
            (Rectangular, Short, 2, Closed, Peaked, (Triangle, 1)),
            (Rectangular, Long, 3, Open, None, (Hexagon, 1)),
            (Rectangular, Short, 2, Open, None, (Circle, 1)) ] = East ;
direction [ (UShaped, Short, 2, Open, None, (Triangle, 1)),
            (BucketShaped, Short, 2, Open, None, (Rectangle, 1)),
            (Rectangular, Short, 2, Closed, Flat, (Circle, 2)) ] = East ;
direction [ (Rectangular, Short, 2, Open, None, (Circle, 1)),
            (Hexagonal, Short, 2, Closed, Flat, (Triangle, 1)),
            (Rectangular, Long, 3, Closed, Flat, (UTriangle, 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 [ (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, (LRectangle, 1)),
            (UShaped, Short, 2, Open, None, (Circle, 1)) ] = West ;
direction [ (BucketShaped, Short, 2, Open, None, (Circle, 1)),
            (Rectangular, Long, 3, Closed, Jagged, (LRectangle, 1)),
            (Rectangular, Short, 2, Open, None, (Rectangle, 1)),
            (BucketShaped, Short, 2, Open, None, (Circle, 1)) ] = West ;

? [ (DoubleRectangular, Short, 2, Open, None, (Triangle, 1)),
    (Rectangular, Long, 3, Closed, Flat, (LRectangle, 1)),
    (Rectangular, Short, 2, Closed, Flat, (Circle, 1)) ] ;
? [ (UShaped, Short, 2, Open, None, (Rectangle, 1)),
    (Rectangular, Long, 2, Open, None, (Rectangle, 2)) ] ;
```

```

%% -- Trans info

import booleans.es ;
import lists.es ;
import systrans.es ;

eq2 : NumWheels -> Bool ;
(eq2 x) = (== x 2) ;

eq3 : NumWheels -> Bool ;
(eq3 x) = (== x 3) ;

projShape : Car -> Shape ;
(projShape (t1,t2,t3,t4,t5,t6)) = t1 ;

projLength : Car -> Length ;
(projLength (t1,t2,t3,t4,t5,t6)) = t2 ;

projNumWheels : Car -> NumWheels ;
(projNumWheels (t1,t2,t3,t4,t5,t6)) = t3 ;

projKind : Car -> Kind ;
(projKind (t1,t2,t3,t4,t5,t6)) = t4 ;

projRoof : Car -> Roof ;
(projRoof (t1,t2,t3,t4,t5,t6)) = t5 ;

projLoad : Car -> Load ;
(projLoad (t1,t2,t3,t4,t5,t6)) = t6 ;

projObject : Load -> Object ;
(projObject (t1,t2)) = t1 ;

projNumObjects : Load -> NumObjects ;
(projNumObjects (t1,t2)) = t2 ;

head : Individual -> Car ;
(head (# x y)) = x ;

last : Individual -> Car ;
(last (# x [])) = x ;
(last (# x y)) = (last y) ;

import sets.es ;

listToSet : Individual -> (Car -> Bool) ;
(listToSet []) = \x.False ;
(listToSet (# y z)) = (union \x.(ite (== x y) True False) (listToSet z)) ;

setExists1 : (alpha -> Bool) -> (alpha -> Bool) -> Bool ;
(setExists1 p t) = \exists x.(&& (t x) (p x)) ;

and2 : (Car -> Bool) -> (Car -> Bool) -> Car -> Bool ;
(and2 p q) = \x.(&& (p x) (q x)) ;

top : alpha -> Bool ;
(top x) = True ;

```

```

%% -- Predicate rewrite system

top >-> listToSet . setExists1 (top) ;
top >-> and2 (top) (top) ;

top >-> projShape . top ;
top >-> projLength . top ;
top >-> projNumWheels . top ;
top >-> projKind . top ;
top >-> projRoof . top ;
top >-> projLoad . top ;
top >-> projObject . top ;
top >-> projNumObjects . top ;

top >-> eqObjectCircle ; top >-> eqObjectHexagon ; top >-> eqObjectSquare ;
top >-> eqObjectRectangle ; top >-> eqObjectLongRectangle ;
top >-> eqObjectTriangle ; top >-> eqObjectUTriangle ;
top >-> eqObjectDiamond ; top >-> eqObjectNull ;

top >-> eqRoofFlat ; top >-> eqRoofJagged ;
top >-> eqRoofPeaked ; top >-> eqRoofCurved ; top >-> eqRoofNone ;

top >-> eqShapeRectangular ; top >-> eqShapeDoubleRectangular ;
top >-> eqShapeUShaped ; top >-> eqShapeBucketShaped ;
top >-> eqShapeHexagonal ; top >-> eqShapeEllipsoidal ;

top >-> eqLengthLong ; top >-> eqLengthShort ;
top >-> eqKindClosed ; top >-> eqKindOpen ;
top >-> eq2 ; top >-> eq3 ;

%%

```

As can be seen, a problem spec file has four parts — data declarations, data sets, universe of transformations (the equivalent of a background theory in relational learning), and predicate rewrite system. These are separated by `%%`. Each statement in the file ends with a semi-colon. Comments are preceded with `--`. The last `%%` token marks the end of the specification. Everything after that is ignored.

Data declarations The data declarations section defines the type of the individuals in the problem setting. Two language constructs are available for this purpose. Data constructors can be defined in the usual way: a list of constants followed by their common signature. Examples include the declarations for *Object*, *Roof*, *Shape*, etc given in the spec file above. Recursive algebraic types can be declared in the usual way. For example, we can declare the list constructors using these two statements.

```

[] : (List a) ;
# : a -> (List a) -> (List a) ;

```

The second language construct allows us to define type synonyms. Examples of these include the declarations of *Car* and *Individual* above.

The system supports the following basic data types: *Bool*, *Int*, *Float*. Product types can be defined using the symbol `*`. See, for example, the declaration of *Car* above. All these language facilities provide users with a convenient way to define complex composite types, of which the declaration of *Individual* is a simple example.

There are two required elements in this section.

- The declaration of type *Individual*. This is the type of the individuals in the learning task.

- The `LEARN` directive. This specifies the name and signature of the function to be learned.

Data sets This section lists the training and test data sets. Training examples have the form $f x = y$, where f is the function declared using the `LEARN` directive, x is a term of type *Individual*, and y is a term of type *Class*. The test data set is specified by commands of the form $? x$, where again x is a term of type *Individual*. The learner will construct a decision tree using only the training set. The induced decision tree is then used to predict the labels of examples in the test data set. The test data set is useful if you want to participate in competitions like the KDD Cup, where there is a real unknown test set that is withheld from the participants initially.

To achieve better modularity, data sets can be defined in a separate file and then imported into the spec file. For example, we can put all the training examples in a file called “trains.es” and write “import trains.es ;” in this section of the spec file.

Universe of transformations This section lists the transformations that are used in the predicate rewrite system. A transformation is declared by providing an identifier, followed by its signature and definition. The identifier can be any alphanumeric string starting with a lower case letter. This is the actual name used in the definition of the predicate rewrite system. Signatures of transformations can be either monomorphic or polymorphic. The declaration of *setExists₁* above is an example of a polymorphic transformation. Parameters (type variables) are alphanumeric strings starting with a lower case letters. Normal types are alphanumeric strings that begins with an upper case letter.

Functions defined in standard Escher libraries like the `booleans` module can be used to define transformations. Simple import statements like “import booleans.es ;” make these available.

The use of polymorphic transformations is discouraged for two reasons. Firstly, the system runs faster when it only has to deal with monomorphic transformations. (The unification algorithm wouldn’t have to work so hard then.) Secondly, polymorphic transformations can cause ‘undesirable’ predicates to be generated if the predicate rewrite system is not defined carefully to constrain the parameters properly.

There is an order defined on the transformations and the order has implications on the regularisation algorithm (see [Llo03, §4.3]). The system orders the transformations in the order they are declared in the specification file. For that reason, *top* must be listed as the last entry in this section. This is checked by the system.

The system comes equipped with some generic transformations that can be readily used.

Predicate Rewrite System The hypothesis search space is defined using a predicate rewrite system, which is simply a collection of predicate rewrites. A predicate rewrite is a statement of the form $p \rightsquigarrow q$, where both p and q are standard predicates. In the default LR predicate enumeration mode, each of p and q must be a regular predicate. This is also checked by the system. The regularity of a predicate is dependent on the order defined on the transformations (see above).

There is one limitation in the current implementation: the head of a predicate rewrite can only be a transformation of rank 0. We see no need to extend this at present, but this can be fixed in future versions if necessary.

The (context-free) grammar for the input specification file is given in Appendix B.

3.2.2 System Output

It’s time to put `ALKEMY` through its paces. Assuming you have set up the system without trouble (see §3.1), the following command issued in the directory `alkemy/SRC` should do.

```
> alkemy ../TUTORIAL/eastwest.spec
```

If everything went well, one should get something similar to the following:

```
Learning Parameters
  Command Line : ../TUTORIAL/eastwest.spec
```

```

Search Strategy      : N/A
Enumeration Strategy : LR
Statistical Test     : N/A
Tree Structure       : tree

```

```

Total predicates tested : 248
Total predicates in FAPtable : 249

```

Decision tree on training set..

direction x =

```

IF listToSet.setExists1 (and2 (projLength.eqLengthShort) (projKind.eqKindClosed)) x THEN
    East    4/4    (4,0)
ELSE
    West    4/4    (0,4)

```

Train Acc = 8/8 (100%)

Predictions on the test set.

```

direction [(DoubleRectangular,Short,2,Open,None,(Triangle,1)),
           (Rectangular,Long,3,Closed,Flat,(LRectangle,1)),
           (Rectangular,Short,2,Closed,Flat,(Circle,1))] = East
direction [(UShaped,Short,2,Open,None,(Rectangle,1)),
           (Rectangular,Long,2,Open,None,(Rectangle,2))] = West
End predictions

```

Total elapsed time (sec) 10.9237

The first part Learning Parameters reports the setting under which the experiment was performed. The induced decision tree is presented in the form of if-then-else rules. Three pieces of information are given for each terminal node: the majority class, the accuracy of the node, and the class distributions. The accuracy of the tree on the training set is given at the end. (In the case of regression-tree learning, the empirical mean of the examples falling in each node and the squared error achieved by the tree are reported instead.) Also reported is the predictions for examples in the test set. Finally, the time spent building the tree is given.

The verbosity option can be set to get a more detailed reporting of the actual search in progress (`alkemy -vv ../EXPERIMENTS/eastwest.spec`). In this mode, a complete trace of the learning process will be sent to the screen. The following is a sample entry:

```

listToSet.(setExists1 (and2 (projLoad.projObject.eqLRectangle) (top)))
1: (1,2) r: (4,3) Ap = 6 Bp = 7 OL = 46 Tested = 203 Best = 8 PRUNED.

```

It reports that the predicate p given in the first line can be used to induce a binary tree as shown in Figure 3. The letters l and r stand for the left and the right subtrees respectively. The 2-tuple

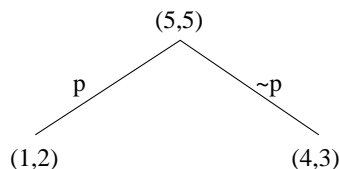


Figure 3: An induced binary tree.

(m, n) indicates that there are m instances in the first class and n instances in the second class. The classes follow the order in which they are declared. In this case, the first class is **East**, the second, **West**.

The system uses a best-first search algorithm, aided by efficient pruning algorithms, to find the best predicate to split a given node. The details can be found in [NL05]. The rest of the

trace entry is related to the accuracy-based heuristic used for classification and the progress of the search algorithm. The number **Ap** is the accuracy of the induced partition, and **Bp** its refinement bound. **OL = 46** means that there are 46 nodes on the open-list. The next field is the total number of predicates examined thus far. **Best** records the current best score. Right at the very end, the system reported that this predicate has been pruned. Interesting predicates are **KEPT**. Also, the best predicates found at any stage are recorded as such with the word **BEST** before the predicate. **EQUAL BEST** predicates are also marked as such.

The very verbose option is useful for debugging. Normally, one is only interested in the major milestones in the learning process. The command `alkemy -v ../TUTORIAL/eastwest.spec` can be used to do that.

3.2.3 Learning Parameters

There are many learning parameters one can set/tune to get different system behaviours. The command `alkemy -h` will produce the full list of parameters supported by the system. I describe here some of the more useful ones.

- **verbosity** (-v) - This sets the level of detail the learner will report its progress.
- **prune** (-p) - This sets the pruning parameter. The value can be a floating point number between 1 and 100. The default is 0. The effect of setting this parameter is to throw away those predicates whose refinement bound is lower than the prune parameter. This pruning mechanism can be used to shorten search time, at the price of incompleteness. The prune parameter is updated if the current best accuracy is larger than its value. In the output, the first number listed for this parameter is the user-specified value. The number in brackets is the final updated prune value. [NL05] contains more information about the behaviour of this parameter.
- **stump** (-n) - This specifies that a decision stump, *i.e.*, a single-split decision tree, is to be learned.
- **cutout** (-c) - This sets the cutout parameter. A search is terminated if the current best accuracy cannot be improved upon within n steps, where n is the cutout value. Further, this parameter is always reinitialised to n upon the discovery of a new best predicate. Setting this can speed up the learning process a lot, at the price of throwing away a potentially large proportion of the search space.
- **cross validation** (-C) - Do an n -fold cross-validation.
- **leave-n%-out** (-t) - Do a leave-n%-out experiment. This can be repeated with a different random test set multiple times by setting the -m option.
- **random seed** (-s) - This number is used to seed the random number generator to produce different random partitioning of the data set into training, validation and test sets.
- **post pruning** (-P) - This specifies that the induced decision tree should be post-pruned. The validation set parameter (-V) should be set accordingly. The default is 10%.
- **enumerate** (-e) - This returns the size of the search space.

There are many other more advanced options. These are left to the user to explore.

3.2.4 System-Generated Transformations

The task of writing the universe of transformations can be a tedious affair, especially for large problems. The implementation alleviates this pain by generating certain transformations automatically. These can be imported from the file “systrans.es” in the universe of transformations section and used in the predicate rewrite system. The rules of generation are as follows, and may be extended in future versions:

- For every constant, an equality transformation is provided. For example, given the declaration `Long, Short : Length`, the system will generate the following two transformations

automatically:

```
eqLengthLong :: Length -> Bool ;
(eqLengthLong x) = (== x Long) ;
eqLengthShort :: Length -> Bool ;
(eqLengthShort x) = (== x Short) ;
```

- For every tuple declaration, the projection transformations are generated. For example, given `Load = Object * NumObjects` ; the system will generate

```
projLoad_0 :: Load -> Object ;
(projLoad_0 (t0,t1)) = t0 ;
projLoad_1 :: Load -> NumObjects ;
(projLoad_1 (t0,t1)) = t1 ;
```

3.2.5 Other Examples

All the examples in [BGCL01] and a few others in the UCI repository have been implemented, and are available for study and experiments. Those spec files can be found in the `EXPERIMENTS` directory.

4 Exercises

Exercise 1. In 1994, Donald Michie *et al.* proposed a new East-West challenge [MMPS94]. A copy of the paper can be found in `alkemy/TUTORIAL/NewEastWest/`. Figures 3.1 and 3.2 show the 20 trains for Competition 1 of the challenge. The first ten came from the original challenge. The remaining were obtained from Stephen Muggleton's random train generator. The first exercise is to use ALKEMY to induce a rule to differentiate between the east-travelling and west-travelling trains. One has the freedom to choose the preferred train representation and the corresponding hypothesis language. The following representation language, used by Muggleton's program to describe the carriages, may be useful.

```
Ellipse, Hexagon, Rectangle, UShaped, Bucket : Shape
Long, Short : Length
NotDouble, Double : Double
None, Flat, Jagged, Peaked, Arc : Roof
Circle, Diamond, Hexagon, Rectangle, Triangle, UTriangle : LoadShape
NumWheels = Int
NumObjects = Int
Load = Object × NumObjects
Car = Shape × Length × Double × Roof × NumWheels × Load
```

A copy of his Prolog representation can be found in the same directory (`20trains.pl`). Note the use of negation to denote west-bound trains. The challenge is to find the simplest solution. It is not hard to find a two-split solution. Can you find a single-split solution?

Exercise 2. The second exercise is intended for those readers who solved the first exercise with the help of Google. ☺ Pick one of the five groups of trains from Competition 3 of [MMPS94] and repeat Exercise 1. The file `100trains.pl` contains the full data set.

Exercise 3. Use ALKEMY to solve the Chess KRK problem. The data set can be found in `alkemy/TUTORIAL/Chess`. The file `lessons2.ps.gz` contains an account of a successful attempt on the problem.

5 Help Needed

We are in the process of building up the library of transformations. If you have implemented a new transformation which you think might be useful for other people, please send it to the author. We are also willing to entertain requests from users to implement new transformations that we deem are interesting and potentially useful.

Bug reports/fixes and suggestions to improve any aspect of the system are always welcome. Please send your email to the same address. Help of any form will be greatly appreciated.

Acknowledgements

This work is supported in part by the CRC for Smart Internet Technologies.

References

- [BD98] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, 1998.
- [BFOS84] Leo Breiman, Jerome Friedman, Richard Olshen, and Charles Stone. *Classification and Regression Trees*. Chapman & Hall, New York, 1984.
- [BGCL01] Antony F. Bowers, Christophe Giraud-Carrier, and John W. Lloyd. A knowledge representation framework for inductive learning. Available at <http://rsise.anu.edu.au/~jw1/>, 2001.
- [Blo98] Hendrik Blockeel. *Top-Down Induction of First Order Logical Decision Trees*. PhD thesis, Departement Computerwetenschappen, Katholieke Universiteit Leuven, 1998.
- [Cla78] Keith Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.
- [FGCL98] Peter A. Flach, Christophe Giraud-Carrier, and John W. Lloyd. Strongly typed inductive concept learning. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446 of *LNAI*, pages 185–194. Springer-Verlag, 1998.
- [Han94] Michael Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
- [Kra96] Stefan Kramer. Structural regression trees. In *Proceedings of the 13th National Conference on Artificial Intelligence*, pages 812–819. AAAI Press, 1996.
- [KW01] Stefan Kramer and Gerhard Widmer. Inducing classification and regression trees in first order logic. In Sašo Džeroski and Nada Lavrač, editors, *Relational Data Mining*, chapter 6. Springer, 2001.
- [Llo95] John W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, Bristol University, 1995.
- [Llo99] John W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3, 1999.
- [Llo03] John W. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data*. Cognitive Technologies. Springer, 2003.
- [MD94] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

- [MMPS94] Donald Michie, Stephen Muggleton, David Page, and Ashwin Srinivasan. To the international computing community: A new east-west challenge. Technical report, Oxford University Computing Laboratory, 1994.
- [Ng05] K.S. Ng. *Learning Comprehensible Theories from Structured Data*. PhD thesis, Computer Sciences Laboratory, The Australian National University, 2005.
- [Ng06] K.S. Ng. *The Alkemy Source Book*. Computer Sciences Laboratory, ANU, 2006. Available at <http://rsise.anu.edu.au/~kee/>.
- [NL05] K.S. Ng and John W. Lloyd. Predicate selection for structural decision trees. In S. Kramer and B. Pfahringer, editors, *Proceedings of the 15th International Conference on Inductive Logic Programming*, LNAI3625, pages 264–278, 2005.
- [NM] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in AI and Logic Programming, Volume 5: Logic Programming*. Oxford.
- [Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- [Qui93] John Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
- [Tho99] Simon Thompson. *Haskell - The Craft of Functional Programming*. Addison-Wesley, 2nd edition, 1999.
- [VD01] Wim Van Laer and Luc De Raedt. How to upgrade propositional learners to first order logic: A case study. In Saso Džeroski and Nada Lavrač, editors, *Relational Data Mining*, chapter 10. Springer, 2001.

A A Short Introduction to Escher

Escher is a functional logic programming language first introduced in [Llo95] and [Llo99]. It was designed with the intention to provide in a simple computational mechanism the best features of functional and logic programming. The basic approach taken in the design of Escher is simple: start from Haskell and add logic programming facilities. (There are other approaches one can take in the design of functional logic programming languages; see, for example, [NM] and [Han94].)

To understand Escher, we need to understand two things. The first is the form of a valid Escher program. The second is the underlying computational mechanism of the language. These are covered in §A.1, which is essentially a summary of [Llo03, Chap. 5]. In §A.2 the relationships between Escher, Haskell and Prolog are clarified. We give some example Escher programs in §A.3.

A.1 Logical Foundation

The logic underlying Escher is a polymorphically typed higher-order logic. The *terms* of the logic are the terms of the typed λ -calculus, formed in the usual way by application, abstraction, and tupling from the set of constants and a set of variables. An Escher program is a theory in the logic in which each formula is a particular kind of equation, namely, a statement.

Definition A.1. A *statement* is a term of the form $h = b$, where h has the form $f t_1 \dots t_n$, $n \geq 0$, for some function f , each free variable in h occurs exactly once in h , and b is type-weaker than h .

The term h is called the *head* and the term b is called the *body* of the statement. The statement is said to be *about* f .

Definition A.2. The definition of a function f is the collection of all statements about f , together with the signature for f .

Definition A.3. An *Escher program* is a collection of definitions.

Example A.1. Here are two Escher programs for performing list concatenations.

$$\begin{aligned} \text{concat}_1 &: \text{List } a \rightarrow \text{List } a \rightarrow \text{List } a \\ \text{concat}_1 \ [] \ x &= x \\ \text{concat}_1 \ (\# \ x \ y) \ z &= (\# \ x \ (\text{concat}_1 \ y \ z)) \\ \text{concat}_2 &: \text{List } a \times \text{List } a \times \text{List } a \rightarrow \Omega \\ \text{concat}_2 \ (u, v, w) &= (u = [] \wedge v = w) \vee \\ &\quad \exists r. \exists x. \exists y. (u = (\# \ r \ x) \wedge w = (\# \ r \ y) \wedge \text{concat}_2 \ (x, v, y)) \end{aligned}$$

The first is written in the functional programming style. (It is in fact a valid Haskell program.) The second is written in the relational or logic programming style. The term $\text{concat}_2 \ (x, y, z)$ evaluates to \top iff z is a concatenation of x and y . We will look at concat_2 in more details to see how logic programming is supported in Escher shortly.

Definition A.4. A *redex* of a term t is a subterm of t that is α -equivalent to an instance of the head of a statement.

Recall that two terms are α -equivalent iff they differ only in the names of bound variables. A subterm s of t is a redex if we can find a statement $h = b$ and a term substitution θ mapping variables to terms such that $h\theta$ is α -equivalent to s .

A redex is outermost if it is not a proper subterm of another redex. Two outermost redexes are by definition disjoint. We are interested primarily in outermost redexes because we want the evaluation strategy to be lazy.

Given an Escher program and a term t , a redex selection rule S maps t to a subset of the set of outermost redexes in t . A standard redex selection rule is the leftmost selection rule S_L . Given a term t , the rule S_L picks out the (single) leftmost outermost redex in t . This is the selection rule implemented in the current Escher interpreter.

Definition A.5. A term s is obtained from a term t by a *computation step* using the selection rule S if the following conditions are satisfied.

1. $S(t) = \{r_i\}$ is non-empty.
2. For each i , the redex r_i is α -equivalent to an instance $h_i\theta_i$ of the head of a statement $h_i = b_i$ for some term substitution θ_i .
3. s is the term obtained from t by replacing each redex r_i by $b_i\theta_i$.

Definition A.6. A *computation* from a term t is a sequence $\{t_i\}_{i=1}^n$ of terms where $t = t_1$ and t_{i+1} is obtained from t_i by a computation step. The term t_1 is called the *goal* of the computation and t_n is called the *answer*.

As is standard in typed declarative languages, run-time type checking is not necessary in Escher. The fact that the body of every statement is type weaker than its head and that every free variable in the head of a statement occurs exactly once ensures that every computation step produces a new term that is well typed.

Central to Escher are some basic functions defined in the booleans module. These functions, together with the term rewriting mechanism described above, provide logic programming facilities in the functional programming setting. I list here some of these boolean functions.

$$\top \wedge x = x \tag{1}$$

$$\perp \wedge x = \perp \tag{2}$$

$$\exists x.\perp = \perp \tag{3}$$

$$\mathbf{u} \wedge \exists x.\mathbf{v} = \exists x.(\mathbf{u} \wedge \mathbf{v}) \tag{4}$$

$$\exists x_1.\exists x_2.\dots.\exists x_n.(\mathbf{x} \wedge (x_1 = \mathbf{u}) \wedge \mathbf{y}) = \exists x_2.\dots.\exists x_n.(\mathbf{x}\{x_1/\mathbf{u}\} \wedge \mathbf{y}\{x_1/\mathbf{u}\}) \tag{5}$$

$$\forall x.(\perp \rightarrow \mathbf{u}) = \top \tag{6}$$

$$\forall x_1.\forall x_2.\dots.\forall x_n.(\mathbf{x} \wedge (x_1 = \mathbf{u}) \wedge \mathbf{y} \rightarrow \mathbf{v}) = \forall x_2.\dots.\forall x_n.(\mathbf{x}\{x_1/\mathbf{u}\} \wedge \mathbf{y}\{x_1/\mathbf{u}\} \rightarrow \mathbf{v}\{x_1/\mathbf{u}\}) \tag{7}$$

Most of these equations are fairly straightforward. One thing is worth noting though. Variables typeset in bold above are actually syntactical variables. So an equation like (4) actually stands for a (possibly infinite) collection of Escher statements with \mathbf{u} and \mathbf{v} instantiated to all possible terms of type boolean. The use of syntactical variables usually come with side conditions. For example, for (4) to be applicable, the syntactical variable \mathbf{u} must not contain a free occurrence of x . Similarly, x_1 must not occur free in \mathbf{u} for (5) and (7) to work.

Example A.2. The following is an example Escher computation using the S_L redex selection rule. The redex selected at each time step is highlighted. Note how Equation (5) given above is used to get rid of the existential quantifiers.

$$\begin{aligned} & \underline{\text{concat}_2([1], [2], w)} \\ &= ([1] = [] \wedge [2] = w) \vee \exists r.\exists x.\exists y.([1] = (\# r x) \wedge w = (\# r y) \wedge \text{concat}_2(x, [2], y)) \\ &= (\perp \wedge [2] = w) \vee \exists r.\exists x.\exists y.([1] = (\# r x) \wedge w = (\# r y) \wedge \text{concat}_2(x, [2], y)) \\ &= \perp \vee \exists r.\exists x.\exists y.([1] = (\# r x) \wedge w = (\# r y) \wedge \text{concat}_2(x, [2], y)) \\ &= \exists r.\exists x.\exists y.([1] = (\# r x) \wedge w = (\# r y) \wedge \text{concat}_2(x, [2], y)) \\ &= \exists r.\exists x.\exists y.(r = 1 \wedge x = [] \wedge w = (\# r y) \wedge \text{concat}_2(x, [2], y)) \\ &= \exists x.\exists y.(x = [] \wedge w = (\# 1 y) \wedge \text{concat}_2(x, [2], y)) \\ &= \exists y.(w = (\# 1 y) \wedge \text{concat}_2([], [2], y)) \\ & \dots \\ &= \exists y.(w = (\# 1 y) \wedge y = [2]) \\ &= (w = [1, 2]) \end{aligned}$$

Example A.3. Given the goal $\text{concat}_2(x, y, [1, 2])$, Escher will return with the following answer

$$(x = [] \wedge y = [1, 2]) \vee (x = [1] \wedge y = [2]) \vee (x = [1, 2] \wedge y = []),$$

which is computed using the same mechanism described in the previous example.

A.2 Escher, Haskell and Prolog

We first explore the relationship between Escher and Haskell. At the logic level, every Haskell program is an Escher program¹, and every Escher program is a (syntactically-correct) Haskell program which may not compile. In that sense, Escher is a superset of Haskell. The difference between Escher and Haskell comes down to the following two points.

- Haskell allows pattern matching only on data constructors. Escher extends this by allowing pattern matching on function symbols as well as data constructors. Examples of equations that Haskell cannot handle but Escher can are those in the `booleans` module given earlier.
- The second thing that Escher can do but Haskell can't is reduction of terms inside lambda abstractions. This mechanism allows Escher to handle sets (and similar data types) in a natural and intensional way. This is usually achieved with the use of syntactical variables.

The extra expressiveness afforded by Escher comes with a price tag, however. Some common optimisation techniques developed for efficient compilation of Haskell code (see [Pey87]) cannot be used in the implementation of Escher. In other words, efficiency is at present still a non-trivial issue for Escher.

We next explore the relationship between Escher and Prolog. Perhaps surprisingly, there is actually a significant overlap between the two languages. In fact, any Prolog program defined without using cuts can be mechanically translated into Escher via Clark's completion algorithm [Cla78]. For example, the Escher program concat_2 given earlier is just the completion of the following Prolog definition.

$$\begin{aligned} \text{concat}_2([], L, L). \\ \text{concat}_2([X|L1], L2, [X|L3]) \leftarrow \text{concat}_2(L1, L2, L3). \end{aligned}$$

Procedurally, there is a difference between Escher and Prolog in that Prolog computes alternative answers one at a time via backtracking whereas Escher returns all alternative answers in a disjunction (a set). This point is illustrated in Example A.3 above.

A.3 Example Programs

I end this short introduction with some example Escher programs. The aim here is to showcase the different styles of declarative programming supported by Escher. An Escher interpreter is available for download as a separate program from <http://rsise.anu.edu.au/~kee>.

Example A.4. Here is how quick sort can be written in Escher. This is just a vanilla Haskell program that doesn't make use of special logic programming facilities in Escher.

$$\begin{aligned} \text{qsort} &: \text{List } a \rightarrow \text{List } a \\ \text{qsort } [] &= [] \\ \text{qsort } (\# x y) &= \text{concat}_1(\text{qsort } (\text{filter } (\leq x) y)) (\# x (\text{qsort } (\text{filter } (> x) y))) \\ \text{filter} &: (a \rightarrow \Omega) \rightarrow \text{List } a \rightarrow \text{List } a \\ \text{filter } p \ [] &= [] \\ \text{filter } p (\# x y) &= \text{if } (p x) \text{ then } (\# x (\text{filter } p y)) \text{ else } (\text{filter } p y) \end{aligned}$$

¹This is not exactly true at the practical level, however, since Haskell has several advanced language features not currently available in Escher. But this is only a software maturity issue. Given more development time, these language features will be implemented and Escher will then be able to run any valid Haskell program.

Example A.5. The following is an example of an Escher program for computing permutations of lists. The function *permute* returns true iff the two input arguments are permutations of each other. The function *delete* is a subsidiary function of *permute* that returns true iff the third argument is the result of removing the first argument from the second argument.

$$\begin{aligned}
& \textit{permute} : (\textit{List } a) \times (\textit{List } a) \rightarrow \Omega \\
& \textit{permute} (\ [], x) = (x = []) \\
& \textit{permute} ((\# x y), w) = \exists u. \exists v. \exists z. (w = (\# u v) \wedge \textit{delete} (u, (\# x y), z) \wedge \textit{permute} (z, v)) \\
& \textit{delete} : a \times (\textit{List } a) \times (\textit{List } a) \rightarrow \Omega \\
& \textit{delete}(x, [], y) = \perp \\
& \textit{delete}(x, (\# y z), w) = (x = y \wedge w = z) \vee \exists v. (w = (\# y v) \wedge \textit{delete} (x, z, v))
\end{aligned}$$

Given *permute* ($[1, 2, 3], [2, 1, 3]$), Escher will return \top . Given *permute* ($[1, 2, 2], x$), Escher will return the answer

$$x = [1, 2, 3] \vee x = [1, 3, 2] \vee x = [2, 1, 3] \vee x = [2, 3, 1] \vee x = [3, 1, 2] \vee x = [3, 2, 1].$$

Example A.6. Here are some standard functions defined on sets.

$$\begin{aligned}
& \textit{union} : (a \rightarrow \Omega) \rightarrow (a \rightarrow \Omega) \rightarrow (a \rightarrow \Omega) \\
& \textit{union } s \ t = \lambda x. ((s \ x) \vee (t \ x)) \\
& \textit{intersect} : (a \rightarrow \Omega) \rightarrow (a \rightarrow \Omega) \rightarrow (a \rightarrow \Omega) \\
& \textit{intersect } s \ t = \lambda x. ((s \ x) \wedge (t \ x)) \\
& \textit{minus} : (a \rightarrow \Omega) \rightarrow (a \rightarrow \Omega) \rightarrow (a \rightarrow \Omega) \\
& \textit{minus } s \ t = \lambda x. ((s \ x) \wedge \neg(t \ x)) \\
& \textit{subset} : (a \rightarrow \Omega) \rightarrow (a \rightarrow \Omega) \rightarrow \Omega \\
& \textit{subset } s \ t = \forall x. ((s \ x) \rightarrow (t \ x))
\end{aligned}$$

Similar functions for multisets can be just as easily defined.

B Syntax

We look at the syntax of the input specification in this appendix. The grammar given below makes use of tokens like `IDENTIFIER1`, `IDENTIFIER2`, `DATA_CONSTRUCTOR`, etc. The regular expression for each of these can be found at the end of this appendix.

As mentioned earlier, an input specification consists of four parts separated by `%%`.

```
input : declaration examples transinfo rewrites "%%" ;
```

In the declaration section, two kinds of type declarations (`typedecls`) are allowed. Data constructors (`constructordecl`) are declared by giving a list of constants (alphanumerics starting with a capital letter) followed by their common signature. Type synonyms (`typedecl`) can be used to give shorter names to types.

The function to be learned (`funcdecl`) is declared in this section as well.

```
declaration : "%%" typedecls funcdecl ;

typedecls : typedecl | typedecls typedecl ;
typedecl : syndecl | constructordecl ;

syndecl : IDENTIFIER2 '=' type ',' ;
constructordecl : dataconstructors ':' type ',' ;

dataconstructors : dataconstructor | dataconstructors ',' dataconstructor ;
dataconstructor : IDENTIFIER2 | DATA_CONSTRUCTOR ;

funcdecl : "Learn" IDENTIFIER1 ':' IDENTIFIER2 "->" codomain ',' ;
codomain : IDENTIFIER2 | "Bool" | "Int" | "Float" ;
```

Training examples and test individuals are declared in the second section. Training examples come in the form of `f x = y`, where `f` is the function being learned, and `y` the label of `x`. Test examples come in the form of `?x`. We can import these from external files as well.

```
examples : "%%" individuals ;

individuals : individual | individuals individual ;

individual : IDENTIFIER1 term '=' label ',' ;
            | '?' term ',' ;
            | "import" FILENAME ',' ;
            ;
label : IDENTIFIER2 | DATA_CONSTRUCTOR | DATA_CONSTRUCTOR_FLOAT ;
```

Transformations used in the predicate rewrite system are declared in the third section. Escher is used to define the transformations and their subsidiary functions. Each `statement` is an Escher statement, possibly with syntactic variables in it. Data constructors can be declared in this section as well.

```
transinfo : "%%" alkemy_statements ;

alkemy_statements : /* empty */ | alkemy_statements alkemy_statement ;
alkemy_statement : statement | typedecl | "import" FILENAME ',' ;

statement : iden ':' type ',' term '=' term ',' ;
iden : IDENTIFIER1 | FUNCTION ;
```

The next section deals with the predicate rewrite system. A predicate rewrite system is just a collection of predicate rewrites. The head of a predicate rewrite is a rank-0 transformation. The tail of a predicate rewrite is a standard predicate. Standard predicates are defined by induction on the declared transformations for the current application.

```

rewrites : "%" rewrite_list ;
rewrite_list : rewrite | rewrite_list rewrite ;

rewrite : IDENTIFIER1 ">->" stdPredicate ';'

stdPredicate : transformation
              | stdPredicate '.' transformation
              | stdPredicate '.' '(' transformation ')'
              | '(' stdPredicate ')'
              ;
transformation : IDENTIFIER1 | IDENTIFIER1 arguments ;

arguments : '(' stdPredicate ')' | '(' stdPredicate ')' arguments ;

```

We next look at the grammar for terms. A `term` is a term possibly with syntactic variables in it. The grammar for `term` is defined inductively as follows. Each syntactic variable is a term. Each variable is a term. Each constant, which can be either a function or a data constructor, is a term. If `t1` and `t2` are terms having appropriate types, then `(t1 t2)` is a term. If `x` is a variable and `t` is a term, then `\x.t` is a term. If `t1, t2, ..., tn` are terms, then `(t1,t2,...,tn)` is a term. Syntactic variables can come with side conditions. These are stated using `sv_condition`. There are four kinds of condition we can state. We can specify that a syntactic variable must be a variable or a constant. We can also require that the instantiation of a syntactic variable be equal or not equal to the instantiation of an earlier syntactic variable.

```

term : SYNTACTIC_VARIABLE | SYNTACTIC_VARIABLE sv_condition
      | VARIABLE
      | FUNCTION | DATA_CONSTRUCTOR | DATA_CONSTRUCTOR_INT
      | DATA_CONSTRUCTOR_FLOAT | DATA_CONSTRUCTOR_STRING
      | IDENTIFIER1 | IDENTIFIER2
      | '(' term term ')'
      | '\ ' VARIABLE '.' term
      | '(' terms_product ')'
      | term_sugar
      ;
terms : term | terms term ;

terms_product : /* empty */ | terms_product ',' term ;

sv_condition : '/' VAR '/' | '/' CONST '/'
              | '/' EQUAL ',' SYNTACTIC_VARIABLE '/'
              | '/' NOTEQUAL ',' SYNTACTIC_VARIABLE '/'
              ;

```

Terms as defined can be cumbersome to work with. To ease the writing of the spec file, syntactic sugars are provided for sets, lists, and the quantifiers. This is how they works.

- A set like `{t1, t2}` will be turned into the term

```
\x.(ite (== x t) True (ite (== x t2) True False))
```

before Escher can process it. Here `ite` is the familiar *if-then-else* function.

- A list like [t1, t2] will be turned into the term (# t1 (# t2 [])).
- In accordance with the mathematics (see [Llo03, pg. 43]), a formula like `\exists x.t` will be turned into the term (`sigma \x.t`) and a formula like `\forall x.t` will be turned into the term (`pi \x.t`).

Another syntactic sugar we provide is the ability to enclose terms obtained from multiple applications within a single pair of brackets. So, a term like `((f x) y) z` can be more simply written as `(f x y z)`.

```
term_sugar : '(' term term terms ')'
           | '{' terms_product '}'
           | '[' terms_product ']'
           | '\ ' "exists" VARIABLE '.' term
           | '\ ' "forall" VARIABLE '.' term
           ;
```

We now look at the grammar for types. A parameter (type variable) is a type. These are alphanumeric characters that start with a lower case letter. Each of the basic nullary type constructors like `Bool`, `Number` and `String` is a type. If `T` is a n -ary type constructor and `t1`, `t2`, ..., `tn` are types, then `(T t1 t2 ... tn)` is a type. If `t1` and `t2` are types, then `t1 -> t2` is a type. If `t1`, `t2`, ..., `tn` are types, then `(t1 * t2 * ... * tn)` is a type.

```
type : IDENTIFIER1
      | "Bool" | "Number" | "String" | IDENTIFIER2
      | '(' IDENTIFIER2 types ')'
      | '(' products ')'
      | type "->" type
      | '(' type ')'
      ;
products : products '*' type | type '*' type ;
types : type | types type ;
```

Here are the regular expressions for the tokens used in the grammar. `IDENTIFIER1` are alphanumeric characters that start with a lower case letter. `IDENTIFIER2` are alphanumeric characters that start with an upper case letter. System-defined data constructors and functions are declared here. A file that can be imported into the spec file must end with ".e". Variables and syntactic variables are also governed by fixed rules here. Care should be taken with variables. A lot of programming errors are associated with the use of variable names that does not actually conform to the grammar.

```
IDENTIFIER1 = [a-z][a-zA-Z0-9\_\\']*
IDENTIFIER2 = [A-Z][a-zA-Z0-9\_\\']*
DATA_CONSTRUCTOR = (True | False | # | [])
DATA_CONSTRUCTOR_FLOAT = -?[0-9]+\.[0-9]+
DATA_CONSTRUCTOR_INT = -?[0-9]+
DATA_CONSTRUCTOR_STRING = \"[a-zA-Z0-9\_\\+\: ]*\
FUNCTION = (== | /= | <= | < | >= | > | && | ||)
FILENAME = [a-zA-Z\0-9\_\\.]+\es
VARIABLE = [m-z][0-9]*
SYNTACTIC_VARIABLE = [a-zA-Z][0-9]*\_SV
```