Using Algebraic Datatypes as Uniform Representation for Structured Data

Markus Mottl (markus@oefai.at)
Austrian Research Institute for Artificial Intelligence
Schottengasse 3
1010 Vienna
Austria
Tel: +43 - (0)1 - 53 36 112 / 19
Fax: +43 - (0)1 - 53 36 112 / 77

Abstract. The question of how to uniformly encode structured data for the purpose of machine learning seems to have been rather neglected so far by researchers in comparison to the abundance of approaches how knowledge could be inferred from certain representations. We therefore propose a well-understood concept from the formal semantics of programming languages as vehicle for the uniform representation of discrete data, namely *algebraic datatypes*. It will be demonstrated in theory and practice that current encodings severely limit the power of widespread machine learning techniques especially what concerns handling of structured information, how algebraic datatypes elegantly extend expressiveness to evade these limitations and that this concept can guide the way to new learning algorithms. As an example, it will be shown how ordinary decision tree learning can be efficiently generalized to this data representation and that the latter provides for an interesting solution both to the missing value problem and to the representation of structured multi-attribute goals. Tight theoretical relations to logic yield valuable insights into complexity and expressiveness.

Keywords: Structured data, Algebraic datatypes, Decision tree learning

1. Introduction

Automation of any kind of problem solving usually requires finding suitable symbolic representations for the problems we intend to solve. Such encodings should be expressive enough to encode any imaginable problem we might have to deal with while imposing as little representational complexity as necessary, both for comprehensibility by humans and straightforward symbol manipulation by machines.

In this work we present a practical example that uncovers limitations of a currently widely-used encoding in the field of machine learning, namely *attribute-value representations*. The extensions we will apply thereupon will lead us to a uniform data representation, which will hopefully provide a suitable basis for the development of more capable general-purpose machine learning systems.

2. Intuitive example: Algebraic datatypes make tasty meals

Let's put ourselves into the role of a restaurant owner who wants to revise his menu and prices to suit the tastes of his typical customers, thus allowing him to maximize his profit by optimizing his customers' satisfaction. His unimaginative cook only knows how to prepare a small variety of *meals* and *drinks*, which he enumerates as follows:

$$\textbf{data}\ Meal\ =\ Pizza \mid WienerSchnitzel \mid TapirSoup$$
$$\textbf{data}\ Drink\ =\ Water \mid Beer \mid Wine$$

The waitress was instructed to ask the customers about their *satisfaction*:

$$\textbf{data}\ Satisfaction\ =\ Low \mid High$$

By gaining knowledge (learning) about the customers' preferences or, in other terms, by finding a predictively most accurate function

$$rate\ ::\ (Meal,\ Drink)\ \rightarrow\ Satisfaction$$

from the set of meals and drinks[1] to the set of satisfaction values, the restaurant owner hopes to improve his business. After having collected some observations regarding customer satisfaction, he leaves this task of finding the explaining function to his computer, which runs a classification program in the spirit of *C4.5*[2]. It can output its results graphically as *decision trees*[3], e.g.:

[1] More precisely: their Cartesian product.

[2] See (Quinlan, 1992).

[3] See e.g. (Mitchell, 1996) for details on ordinary decision tree learning and attribute-value representation of data.

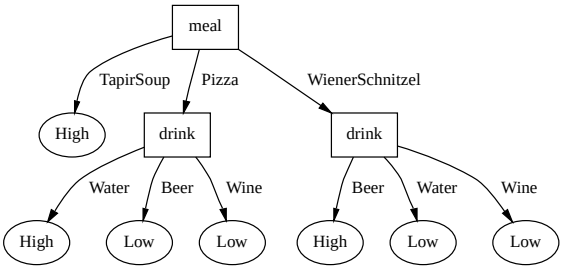

Figure 1. Decision tree (graphical representation)

Or as semantically equivalent text representation using *case-statements*:

case *meal* **of**
  TapirSoup → *High*
  Pizza →
    case *drink* **of**
      Water → *High*
      Beer → *Low*
      Wine → *Low*
  WienerSchnitzel →
    case *drink* **of**
      Water → *Low*
      Beer → *High*
      Wine → *Low*

We will use textual representations as given above[4] for the rest of this paper.

2.1. Cooking indistinguishable food can make meals uneatable

The restaurant owner's greediness makes him ask for even more accurate predictions of customer satisfaction. From his cook he learns that it may well depend on additional attributes *associated with each kind of meal*, for example the diameter of the *WienerSchnitzel*, the kind of *PizzaTopping* or whether the *TapirSoup* is spiced or not.

How can we add this information in traditional representations as used e.g. by most decision tree learners and related approaches? These algorithms usually only expect a fixed number of attributes with a

[4] The educated reader may have already realized that we are actually using the purely functional, lazy language *Haskell* for representation. This may allow us to exploit the powerful mathematical tool of *equational reasoning* to analyze and transform learnt models. More details can be found here: http://www.haskell.org

certain number of tags standing for discrete values, but in our case this may not be enough. One could imagine the following workarounds:

1. Adding an attribute *PizzaTopping* is certainly possible, e.g.:

   data *PizzaTopping* = *Cheese* | *Tomatoes* | *Mushrooms*

   We would then have to revise the type of the function we want to learn to:

   rate :: (*Meal*, *PizzaTopping*, *Drink*) → *Satisfaction*

   But unfortunately, this would allow rather non-sensical models like:

   case *meal* **of**
   TapirSoup → *High*
   Pizza → *Low*
   WienerSchnitzel →
   case *pizza_topping* **of**
   Cheese → *Low*
   Tomatoes → *Low*
   Mushrooms → *High*

   Surely, nobody would seriously intend to eat a *WienerSchnitzel* with a *PizzaTopping*: a rather uneatable combination.

2. Another solution attempt to get the menu recipes right could be to replace the tag *Pizza* with the tags *CheesePizza*, *TomatoePizza* and *MushroomPizza*. This approach is inappropriate for various reasons:

   – It is generally infeasible due to combinatorial explosion of the number of tags if there is more than one sub-attribute associated with some types of values: all possible combinations of associated tags would have to be added. Who would seriously want to work with thousands of tags of the sort *TunaCheeseTomatoesSalamiPizza*?
   – There would not be any obvious solution at all if we consider associated attributes that come from infinite sets, e.g. continuous numeric variables like the diameter of our *WienerSchnitzel* or, as we will see further below, infinite sets of discrete values that are constructed inductively. This would require us to do the impossible, to create an infinite number of distinct tags. In the case of real numbers this would even be an uncountably large set.

   - Another limitation is that we cannot always create simple models from data that contains many irrelevant associated attributes and few relevant ones. The model would have to consider all tags that are formed of the relevant attributes, e.g. all meal tags containing *Pizza* and *Tuna*. This would lead to exceedingly large and greatly redundant models, because we would get equivalent model parts for all tag combinations that meet this criterion — a possibly huge number considering combinatorial explosion of the number of tags.

3. Still another workaround suggests the introduction of an additional boolean attribute for each associated attribute to indicate whether it is relevant in a certain context or not. This leads to more redundant data, because irrelevant cases require a dummy value for the associated attribute. Furthermore, the machine learning system would require ways of figuring out when some attribute is actually relevant for classification, which is a waste of computation time.

The best we can do is therefore not to pretend that information can be represented with atomic elements (value tags, numeric variables) alone: we need ways to cope with *structured data values*.

2.2. Spicing meals with algebraic datatypes

Algebraic datatypes have become very widespread in modern functional and logic programming languages (e.g. Haskell, ML, Mercury, etc.) because of their nice formal properties. An introduction to the formal semantics of programming languages, including a thorough formal treatment of algebraic datatypes can be found in (Winskel, 1993). We are highly convinced that modern developments in the field of formal semantics can be fruitfully applied to machine learning, too, as our work attempts to show. Previous attempts of introducing such concepts can be traced to e.g. (Flach, 2000) and (Bowers et al., 2000), who try to integrate them with inductive logic and functional programming (ILP and IFP).

In contrast to their work we focus on algebraic datatypes, which have the convenient property that their values can only be constructed in a unique, unambiguous way. Thus, we can derive explicit notions of information contents and entropy for them in later parts of this work. The generalization of decision tree learning, which we will later describe, is essentially a step towards inductive functional programming (IFP), but the generalized data representation itself is oblivious to the style of induction as it is to the general-purpose programming languages

mentioned above. It may therefore as well be usefully applied within the framework of inductive logic programming (ILP).

2.2.1. *Meals that make a difference*

The first concept necessary for structuring information is a facility that allows us to make a difference between values we are concerned about, i.e. being able to partition the set of all values into subsets. Without being able to distinguish between values, we could never gain information: all would be the same to us.

We can represent this partitioning by labeling values that belong to a specific domain with a unique name (tag) or, in other terms, by forming a *discriminated union*[5] of the partitions of values in question. This indispensable concept is necessarily present in all representations, though most often in a limited form.

The limitation that often applies as, for example, in the mentioned attribute-value representation that is commonly used in data-mining, is that the discriminated union is only allowed for atomic values, i.e. ones that cannot be further described. These atomic values are exactly identified by their tag alone. E.g. we only say *Pizza* if it is not of interest or cannot be known whether the value in question has further attributes like *PizzaTopping* that distinguish it from others.

If, however, the value can be distinguished from others that otherwise share some property, we will have to mention the distinguishing factors. For instance, we might want to revise our representation of meals as follows:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Meal} = \mathit{Pizza}\ \mathit{PizzaTopping} \\
&\qquad\qquad \mid\ \mathit{WienerSchnitzel}\ \mathit{Diameter} \\
&\qquad\qquad \mid\ \mathit{TapirSoup}\ \mathit{Spiced} \\
&\textbf{data}\ \mathit{PizzaTopping} = \mathit{Cheese} \mid \mathit{Tomatoes} \mid \mathit{Mushrooms} \\
&\textbf{type}\ \mathit{Diameter} \quad = \mathit{Float} \\
&\textbf{type}\ \mathit{Spiced} \quad\ = \mathit{Bool} \\
&\textbf{data}\ \mathit{Bool} \quad\ = \mathit{False} \mid \mathit{True}
\end{aligned}
$$

Each of the tags *Pizza*, *WienerSchnitzel* and *TapirSoup* labels a value that is described by another attribute (another set of values). This way we can discriminate between meals more precisely. This other attribute may again be a discriminated union of sets (types) of values as in the data definition of *PizzaTopping* or equivalent[6] to some other

[5] Also sometimes referred to as *sum type*.

[6] The keyword **type** can be used to introduce type synonyms for readability. For example, *Spiced* is more intuitive than just saying *Bool* for boolean values.

type, e.g. the floating point numbers *Float*[7]. The following are examples of values legal with respect to our upper specification:

Pizza Mushrooms
WienerSchnitzel 23.7
TapirSoup True

Note that we can refine attributes even more deeply as with the following alternative definitions:

data *PizzaTopping* = *Cheese CheeseSort* | *Tomatoes* | *Mushrooms*
data *CheeseSort* = *Mozzarella* | *Gorgonzola*

With this definition an example for a legal value would be:

Pizza (*Cheese Mozzarella*)

Such nested discriminated unions also make data specifications much more modular: we can refine any part without having to change any of the other definitions as we would have to when applying the clumsy tag replacement workaround mentioned previously.

2.2.2. *Products of ingredients allow for varied meals*
Because we will often need more than one attribute to specify the detailed properties of some value, we will also require the concept of (Cartesian) *products of sets* (types) of values[8]. For example, the diameter of a *Pizza* may also yield valuable information about customer satisfaction. Thus, we could define meals as follows (the other definitions remain the same):

data *Meal* = *Pizza Diameter PizzaTopping*
| *WienerSchnitzel Diameter*
| *TapirSoup Spiced*

A "legal" meal could then be:

Pizza 22.86 (*Cheese Gorgonzola*)

Such products of types (here of *Diameter* and *PizzaTopping*) greatly increase the number of degrees of freedom we have to describe some value (e.g. some meal).

[7] We will ignore handling of non-discrete (e.g. numeric) data in this paper, though this is can be easily integrated into the new concept.
[8] Algebraic datatypes are also often referred to as *polynomial* or *sum-of-product types*.

If we take a look at the combination of the two previously introduced concepts (sums and products), we notice that the commonly used representation of data, namely the attribute-value representation, is just a special case: ordinary decision tree learners require data represented as a product of non-structured sums, the product being the Cartesian product of all attributes (variables), the sum being used to collect values into the various attributes. Therefore, our proposal, which allows sums and products at any level to create structured values, is a true generalization of the more common representation.

What we would like to know now is how models that map between such data representations can look like, i.e. what kind of functions an extended machine learning system could induce. Let's take this example:

```
case meal of
  TapirSoup _ → High
  Pizza diameter topping →
    if diameter < 10 then
      Low
    else
      case topping of
        Cheese cheese_sort →
          case cheese_sort of
            Mozzarella → Low
            Gorgonzola → High
        Tomatoes → Low
        Mushrooms → High
  WienerSchnitzel _ →
    case drink of
      Water → Low
      Beer  → High
      Wine  → Low
```

We consider the case when a *Pizza* is served to explain how this model should be interpreted: its associated attributes *Diameter* and *PizzaTopping* are *bound* to the variables *diameter* and *topping* respectively. The right hand side of this case may now use these variables if required. Here, the diameter is compared to the value 10 in a conditional expression. If the pizza is too small, the resulting value of type *Satisfaction* will be *Low*. Otherwise it depends on the *PizzaTopping*, whether the customer will be satisfied. In the case of a *Cheese*-topping, we will also have to consider what sort of cheese we are observing.

Sometimes an associated attribute may be irrelevant to explain customer satisfaction. In our example, one can see this in the case of

TapirSoup and *WienerSchnitzel*: their associated attributes are not considered, which is indicated by the dummy variable "_", which can be used if some value should not be bound to an identifier.

The main concept behind the interpretation of our model is usually called *pattern matching*, because the data which we are analyzing is matched against patterns that contain variables. The henceforth bound variables can then be used on the right hand side of patterns for further computations. As can be hopefully seen, these kinds of models are an extremely expressive vehicle for describing functional relations between data structures.

2.3. Unbounded customer satisfaction with inductively defined data structures

Though algebraic datatypes (sum of products types) as presented so far are already fairly expressive, there is another simple concept that can be applied to increase expressiveness — even infinitely!

Our restaurant owner's waitress might come to the conclusion that a simple-minded classification of customer satisfaction into *Low* and *High* may not be sufficient to reflect reality appropriately. What is missing so far is the possibility to speak about various degrees of satisfaction. Assuming that human satisfaction can be unbounded in both directions and given that people usually do not speak about their satisfaction in numeric terms, we need a discrete representation that can encode structured values of unlimited size.

The clever waitress therefore advises her boss to revise the definition of *Satisfaction* as follows:

$$\textbf{data}\ \mathit{Satisfaction} = \mathit{Low} \mid \mathit{High} \mid \mathit{Very\ Satisfaction}$$

This definition differs from all previously presented ones as it mentions its type constructor[9] *Satisfaction* not only on the left but also on the right hand side. This still leaves *Low* and *High* as values of satisfaction, but also allows for unbounded degrees like *Very Low* or *Very* (*Very High*), etc. Achieving this kind of unbounded expressiveness with usual encodings is impossible as was argued in section 2.1. Note that mutually recursive definitions of datatypes are also possible, i.e. the type constructor of one definition appears on the right hand side of some other definition and vice versa.

As a side note, it should be pointed out that there are always only finitely many models (functions) for non-recursive data structures, but (usually) infinitely many for recursive ones. More precisely, recursive

[9] The name given to some set of values.

data structures on the right hand side and non-recursive ones on the left hand side of a classification task usually imply a countable infinity of possible models, while in the opposite case there are even uncountably many ones.

Search space sizes for problems can be easily calculated by treating the sum operator in type definitions as arithmetic sum, the product as arithmetic product and the function type constructor (the "arrow") as exponentiation, while data constructors are interpreted as the unit element. This is useful for getting an idea about the complexity of the learning task.

2.4. Partial knowledge of tastes

Since our new definition of *Satisfaction* also imposes *structure on result (class) values*, it may be interesting to point out another property we can exploit for machine learning: it is possible to only partially predict result values!

Let's assume that exotic meals like *TapirSoup* will lead to "very" different degrees of satisfaction among customers. Some will like the exotic taste a lot, others might feel great disgust. So the satisfaction value might be either *Very High* or *Very Low*, but the argument to *Very* may not be clear in advance. Some machine learning system could therefore indicate to the user that the potential outcome of consumption will be extreme by predicting *Very* followed by a (possibly more biased) classification of the remaining structure. This may yield valuable information for risk-averse restaurant owners who dislike extremes and prefer easily predictable customer satisfaction. The commonly used atomic class values would not allow us to explicitly represent this kind of knowledge in a model.

Of course, this partial classification also applies to products of values: some parts of a product may already be uniquely predictable, whereas others still require further learning to be distinguishable.

An example combining both aspects of partial classification would be the following, where a pair (Cartesian product) of satisfaction values (e.g. for a couple having dinner) are predicted simultaneously:

```
let
  v1 =
    case drink of
      Water → Low
      Beer  → High
      Wine  → High
in
(Very v1, Low)
```

This might indicate to the restaurant owner that men (left element of the pair) always tend to extreme levels of satisfaction, which, however, depend on the kind of drink, whereas their spouse would be mildly discontent in any case.

3. Generalizing Decision Tree Learning to Algebraic Datatypes

In this section we will show that ordinary decision tree learning can be cleanly and efficiently generalized such that instead of only handling usual attribute-value representations it can operate on arbitrarily structured, even recursive input and output data. We call this algorithm[10] *AIFAD*[11] (Automated Induction of Functions over Algebraic Datatypes). Note, however, that this generalization will not allow us to induce recursive functions, which is still an open but extremely interesting research topic.

3.1. Description of the generalized algorithm by example

Assume that we have a product of sum types on both the left- and right hand side of a classification task, e.g.:

$$(\mathit{Meal},\ \mathit{Drink}) \rightarrow (\mathit{Satisfaction},\ \mathit{Satisfaction})$$

where the corresponding type constructors are defined as follows:

$$\begin{aligned}
&\textbf{data}\ \mathit{Meal} = \mathit{Pizza}\ \mathit{PizzaTopping}\\
&\qquad\qquad \mid\ \mathit{WienerSchnitzel}\ \mathit{Diameter}\\
&\qquad\qquad \mid\ \mathit{TapirSoup}\ \mathit{Spiced}\\
&\textbf{data}\ \mathit{PizzaTopping} = \mathit{Cheese} \mid \mathit{Tomatoes} \mid \mathit{Mushrooms}\\
&\textbf{data}\ \mathit{Diameter} \quad\ = \mathit{Large} \mid \mathit{Small}\\
&\textbf{data}\ \mathit{Spiced} \quad\ \ = \mathit{False} \mid \mathit{True}
\end{aligned}$$

Let's consider the following dataset of input-output mappings that we want to learn:

[10] A more formal representation of this algorithm will be available in an upcoming, longer version of this paper, which will appear as technical report at: http://www.oefai.at/oefai/tr-online

[11] A fairly fully-featured implementation, which can handle industrial size data efficiently, can be found here: http://www.oefai.at/~markus/aifad

$$
\begin{array}{ll}
(\mathit{WienerSchnitzel\ Large},\ \mathit{Water}) & \rightarrow (\mathit{Very\ Low},\ \mathit{High}). \\
(\mathit{WienerSchnitzel\ Small},\ \mathit{Beer}) & \rightarrow (\mathit{Very\ Low},\ \mathit{Low}). \\
(\mathit{TapirSoup\ True},\ \mathit{Beer}) & \rightarrow (\mathit{Very\ Low},\ \mathit{High}). \\
(\mathit{TapirSoup\ False},\ \mathit{Beer}) & \rightarrow (\mathit{Very\ High},\ \mathit{High}).
\end{array}
$$

3.1.1. *Normalization of data on the right hand side (output)*
The first thing we can do is look out for (parts of) structures on the right hand side and see if they are unique for all samples. One of the following three scenarios can happen:

1. All elements on the right hand side are identical. In this case we can create a leaf node, i.e. just return this value. The algorithm could then continue learning the decision tree for other branches that have not been done yet or return the final model otherwise.

2. Samples share common structures but they are not completely identical. In this case we factor out the identical parts by using a *let-abstraction* and continue partitioning the factorized samples by choosing an input variable for splitting. If there are no such variables left, the most likely value will have to be estimated (see section 3.3).

3. Samples do not share common structure. In this case we will have to look at our input variables to discriminate among them or compute a most likely value again if no more input data is available.

In our example dataset case two applies: the topmost constructor of all first satisfaction values is unique. Thus, we can already construct part of the model:

$$
\begin{array}{l}
\textbf{let} \\
\quad (v1,\ v2)\ =\ \ldots \\
\textbf{in} \\
(\mathit{Very}\ v1,\ v2)
\end{array}
$$

The factorized set of samples now looks as follows:

$$
\begin{array}{ll}
(\mathit{WienerSchnitzel\ Large},\ \mathit{Water}) & \rightarrow (\mathit{Low},\ \mathit{High}). \\
(\mathit{WienerSchnitzel\ Small},\ \mathit{Beer}) & \rightarrow (\mathit{Low},\ \mathit{Low}). \\
(\mathit{TapirSoup\ True},\ \mathit{Beer}) & \rightarrow (\mathit{Low},\ \mathit{High}). \\
(\mathit{TapirSoup\ False},\ \mathit{Beer}) & \rightarrow (\mathit{High},\ \mathit{High}).
\end{array}
$$

3.1.2. *Normalization of data on the left hand side (input)*
Since we are not done yet, we will have to choose an input variable to discriminate further. Before we can do this, we will have to get rid

of redundant information by factoring out common structure again, thus leaving only input variables whose topmost constructors are not unique. In our example above this is already the case. If, instead, we had the following input data:

(*WienerSchnitzel Large*, *Water*).
(*WienerSchnitzel Large*, *Beer*).
(*WienerSchnitzel Small*, *Beer*).
(*WienerSchnitzel Small*, *Beer*).

Then we would have to factorize it such that the data constructor *WienerSchnitzel* is dropped, revealing the non-redundant diameter information beneath it. Of course, this process would have to be repeated recursively if sub-attributes were redundant, too. Some old variables may completely vanish and arbitrarily many new ones may enter the game depending on the structure of the input data.

If new variables get added, we will have to record for each of them, which variables (and possibly sub-attributes) have to be matched in order to access the new variables. In the upper case this would mean recording for the freshly added diameter variable that we have to match the meal variable first to make the diameter of the *WienerSchnitzel* available. Of course, the match case would only contain two branches then: one for *WienerSchnitzel* whereas the other would have to be taken in any other case, since we do not have any concrete samples for refining this partition. The match would then look as follows (the underscore being a catch-all dummy variable):

case *meal* **of**
  WienerSchnitzel diameter → ...
  _ → ...

It might be tempting to think that we could immediately add a pattern match to the model to get at the non-redundant constructors. But this is a bad idea for several reasons:

- It may happen that the revealed information (the new variable) is never actually used (here: that the diameter is not required for classification). In this case we would have to undo the match.
- Even if the variable is used, it may not be required in all branches of the model tree beneath the pattern matching. This would imply that the program "asks questions" whose answer is not always needed. Just imagine an expert system for medical diagnosis that asked the doctor to perform surgery on your stomach to gain more medical data although you are only concerned about your

headache. And all this, just because this information is indeed useful in some extremely rare cases of illness (rare branches in the decision tree used for diagnosis).

The classification process should be driven as far as possible by available data and at the same time require as little data as possible to give a result, which would be violated here.

– The model complexity would increase, because redundant decisions (discrimination of the data) would have to be taken earlier than necessary and thus apply to more samples. Hence, the average number of input bits in the training data consumed to predict an output bit in the same would increase. This is bad if we want to compare models in terms of such a complexity criterion measuring the compression of data.

Therefore, the optimum method to destruct (match) input data and to construct output data is by matching (redundant) input data as *late* as possible down the model tree while constructing (redundant) output data as *early* as possible up the tree. This is, why we have to delay matching by remembering for newly added variables, which matches still need to be performed directly in advance.

If we now assume that indeed the diameter had been chosen for partitioning the training data, the model would be constructed like this:

```
case meal of
  WienerSchnitzel diameter →
    case diameter of
      Large → ...
      Small → ...
  _                        → ...
```

This guarantees that *meal* is matched as late as possible, i.e. when *diameter* is really needed.

3.2. Entropy measures and choosing input variables for partitioning the dataset

Decision tree learning as performed by e.g. ID3 or C4.5 uses greedy heuristics for choosing variables that build on the concept of statistical *entropy*[12]. The idea is to consume (match) as little information as possible on the left hand side of the samples while gaining (producing, constructing) as much information as possible on the right hand

[12] See (Shannon and Weaver, 1949) on the mathematical theory behind this concept.

side. To do this we will first have to generalize the common notions of entropy (average information contents) from attribute-value representations to algebraic datatypes. Though, formally speaking, there is only one statistically correct entropy measure, we will see that several other reasonable entropy-like measures can be designed, all of which have different properties and trade-offs.

For one non-structured variable X containing N_X elements that come from classes[13] $C_1, \ldots, C_k$, entropy in bits $\mathrm{H}(X)$ is defined as:

$$\mathrm{H}(X) = -\sum_{c=1}^{\mathrm{classes}(X)} \mathrm{p}_X(c) \times \log_2(\mathrm{p}_X(c))$$

where function "classes" returns the number of constructors used to form the discriminated union of variable X, and $\mathrm{p}_X(c)$ is the relative frequency (probability) of class c in variable X ($\mathrm{card}_X(c)$ returns the number of values in variable X starting with constructor c):

$$\mathrm{p}_X(c) = \frac{\mathrm{card}_X(c)}{N_X}$$

As a side note, calculation of entropy can be rearranged for slightly more efficient computation as follows if the class histogram is known:

$$\mathrm{H}(X) = \log_2(N_X) - \frac{\sum_{c=1}^{\mathrm{classes}(X)} \mathrm{card}_X(c) \times \log_2(\mathrm{card}_X(c))}{N_X}$$

3.2.1. *Entropy of products of variables*
If we want to compute the joint (dependent) entropy of a *product* of n variables $X_1, \ldots, X_n$, the corresponding formula is:

$$\mathrm{H_d}(X_1, \ldots, X_n) = \sum_{i=1}^{n} \mathrm{H}(X_i | X_{i-1}, \ldots, X_1))$$

More pragmatically speaking, we first compute the entropy for the first variable. Then we split up the remaining variables into partitions depending on the class values of the first variable and recurse for each of those. The returned entropies of these partitions are weighted by their size and summed up:

$$\mathrm{H_d}(X_1, \ldots, X_n) = \mathrm{H}(X_1) + \sum_{c=1}^{\mathrm{classes}(X_1)} \mathrm{p}_{X_1}(c) \times \mathrm{H_d}(X_2, \ldots, X_n | X_1 = c)$$

[13] Alternatives in the corresponding sum type.

We call the above entropy formula *dependent entropy*. If the variables $X_1, \ldots, X_n$ are independent of each other, i.e. the partitions are equally large at each split, the joint (independent) entropy reduces to:

$$\mathrm{H_i}(X_1, \ldots, X_n) = \sum_{i=1}^{n} \mathrm{H}(X_i)$$

If we happen to know that our data has this property or that existing dependencies are negligible, we can exploit this to greatly improve performance of entropy calculation, which is one of the most expensive parts of the whole algorithm. Therefore, it may sometimes be beneficial to use the above formula instead: we call this *independent entropy*.

3.2.2. *Entropy of structured variables*
Now we will consider variables that have substructure and call this *deep entropy*. If we do not do this, i.e. if we only consider topmost constructors of values as above, then we refer to *shallow entropy*. Shallow entropy is useful if sub-variables are (almost) identical for all values having a specific constructor, i.e. contribute little to nothing to the total entropy.

There is actually only a small difference to the calculation of shallow entropy as described in the previous subsection: whenever we compute the entropy of one variable, i.e. when we "recognize" its values, then we may add sub-variables associated with each constructor to the partitions of values arising from the entropy calculation. For dependent, deep entropy calculation this is formally expressed as follows:

$$\begin{aligned}&\mathrm{H_{dd}}(X_1, \ldots, X_n) = \\ &\quad \mathrm{H}(X_1) \\ &\quad + \sum_{c=1}^{\mathrm{classes}(X_1)} \mathrm{p}_{X_1}(c) \times \mathrm{H_{dd}}(X^1_{1|c}, \ldots, X^{\mathrm{svars}(c)}_{1|c}, X_2, \ldots, X_n | X_1 = c)\end{aligned}$$

where $\mathrm{svars}(c)$ returns the number of sub-variables for constructor c, and $X^n_{1|c}$ is the n-th sub-variable of variable X_1 limited to values tagged with constructor c.

We may again define an independent version of deep entropy calculation:

$$\mathrm{H_{id}}(X_1, \ldots, X_n) = \sum_{i=1}^{n} \sum_{c=1}^{\mathrm{classes}(X_i)} \mathrm{p}_{X_i}(c) \times \mathrm{H_{id}}(X^1_{i|c}, \ldots, X^{\mathrm{svars}(c)}_{i|c})$$

Thus, we have finally defined four measures useful for determining average information contents, the deep, dependent entropy being the statistically exact one for algebraic datatypes.

3.2.3. *Choosing variables for partitioning*
Having a measure for average information contents, we can apply the usual heuristics as found in ID3 or C4.5 to select an input variable for partitioning datasets. The *information gain* is the number of bits gained by partitioning a dataset on an input variable and is defined as follows:

$$\begin{aligned}&\mathrm{Gain}_A(X_1,\ldots,X_n) = \\ &\quad H_{dd}(X_1,\ldots,X_n) - \sum_{c=1}^{\mathrm{classes}(A)} \mathrm{p}_A(c) \times H_{dd}(X_1,\ldots,X_n|A=c)\end{aligned}$$

Of course, this gain could be defined analogously using a different information measure. ID3 applies the gain criterion by selecting the input variable that boasts the highest gain.

The *gain ratio* expresses the number of bits gained divided by the number of bits consumed by using a certain input variable for partitioning. The variable having the highest gain ratio is considered to be best:

$$\mathrm{GainRatio}_A(X_1,\ldots,X_n) = \frac{\mathrm{Gain}_A(X_1,\ldots,X_n)}{\mathrm{H}(A)}$$

Since possible sub-variables associated with variable A remain in the set of input variables, only the shallow entropy $\mathrm{H}(A)$ is required here.[14].

[14] Side note: C4.5 uses the slightly adapted *gain ratio criterion*: it first selects input variables that have at least average gain and chooses the one of those having the highest gain ratio. Unfortunately, the rationale behind this idea as explained in (Quinlan, 1986) and repeated in other publications (e.g. (Mitchell, 1996)) is mathematically not justified: they claim that as $\mathrm{H}(A)$ becomes smaller, the gain ratio becomes larger. Looking at the fraction, this may seem right at first sight, but is invalid, because the information gain is dependent on $\mathrm{H}(A)$: variables that contain little information are less likely to evenly partition other data, which generally implies smaller gains. The empirical results concerning reduced number of nodes in decision trees can be easily explained theoretically, but do absolutely not imply anything about predictive accuracy. The claim that predictive accuracy can be higher in certain constructed cases still allows that it could be smaller in others. Experiments on 32 datasets of real data that we have performed did not show any mentionable advantage for either gain ratio or the modified criterion.

Getting back to our example dataset, we now demonstrate how to choose one of the two visible input variables. The right hand side of the dataset is:

$(Low,\ High).$
$(Low,\ Low).$
$(Low,\ High).$
$(High,\ High).$

Dependent entropy gives us a value of 1.5 bits, whereas independent entropy would give us 1.62 bits[15]. Splitting the dataset using variable *meal* would result in the following two partitions, the first one for meal values *WienerSchnitzel*, the second one for *TapirSoup*[16]:

$(Large,\ Water) \rightarrow (Low,\ High).$
$(Small,\ Beer) \rightarrow (Low,\ Low).$

$(True,\ Beer) \rightarrow (Low,\ High).$
$(False,\ Beer) \rightarrow (High,\ High).$

Entropy would now decrease to 1 bit, irrespective of whether we used the dependent or independent formula. Therefore, the information gain would be 0.5 bits for dependent entropy and 0.62 bits for independent entropy. Since the chosen input variable *meal* contains 2 bits of information in the matched (topmost) constructor, the dependent gain ratio is 0.25, the independent one 0.31.

If we had chosen input variable *drink* instead, the following partitions would arise, the first one for drink *Water*, the second one for *Beer*:

$WienerSchnitzel\ Large \rightarrow (Low,\ High).$

$WienerSchnitzel\ Small \rightarrow (Low,\ Low).$
$TapirSoup\ True \rightarrow (Low,\ High).$
$TapirSoup\ False \rightarrow (High,\ High).$

Here we would get dependent entropy of 1.19 bits and independent entropy of 1.38 bits [17], leading to a dependent information gain of 0.31 and an independent one of 0.25. Since the information consumed by choosing variable *drink* is 0.81 bits, the dependent gain ratio would

[15] All numbers rounded to two digits after the decimal point.
[16] Note, how different types of sub-variables are added to the input variables of each partition!
[17] Note that $\forall X.\mathrm{H_d}(X) \leq \mathrm{H_i}(X)$.

therefore be 0.31/0.81 = 0.38, the independent one 0.25/0.81 = 0.30 (rounded).

Taking dependent gain ratio as selection criterion, we would favor variable *drink* (0.38) over *meal* (0.25). We will continue with the choice using dependent entropy, but would like to point out here that independent entropy would have chosen otherwise, since there variable *drink* has a gain ratio of 0.30 whereas the one for *meal* would have been 0.31. Our choice yields the following, still incomplete model:

```
let
  (v1, v2) =
    case drink of
      Water → ...
      Beer → ...
      _ → ...
in
(Very v1, v2)
```

Note that we also need to learn a model for cases when some specified kind of *drink* has never been observed before, as happens above for *Wine*. The unobserved constructors are omitted and a dummy binding "_" is used instead to indicate this. Common decision tree learning now simply estimates the most likely value for the default case[18].

3.3. Computing the most likely value

Continuing in the default branch of the previous model, we will show in this subsection how the most likely value of output data can be calculated. Let's consider the following example data:

(*Low*, *Low*).
(*Low*, *Very High*).
(*High*, *High*).
(*Very High*, *High*).

If we only considered each variable separately when choosing the most frequent element as estimate for the most likely one, we would choose the constructor *Low* for the first variable and the constructor *High* for the second, both of which occur twice. Thus, our estimate for the most likely value for both variables would be (*Low*, *High*). But interestingly, this estimate is itself not a value of the example data!

[18] It is possible to continue learning in default branches by continuing with the dataset before the split but using the remaining variables for learning instead. This, however, raises some interesting questions concerning computational complexity that we do not attempt to answer here.

Therefore, calculating the most likely value independently is an adventurous method. The machine learning system might predict observations that it has actually never seen before, which is a kind of inductive generalization that has not previously existed in traditional decision tree learning. This property can be both an advantage and a disadvantage: on the one hand, the system might benefit from this new generalization capability in terms of accuracy while allowing very fast estimation of this value. On the other hand, it may construct values in such a way that they do not actually make sense. If the latter can lead to harmful classifications, it's better to not use this feature and apply dependent calculation of the most likely value instead.

How do we calculate the most likely value dependently? There are several useful ways of doing it. One is that we assume that all values are equally weighted and that we pick the one that occurs most frequently, i.e. it must be absolutely equivalent to other values. This, however, is not very useful when we have to deal with many variables on the right hand side: it would be extremely unlikely that two values are equivalent, making all choices of observed values possible. On the other hand, this most likely value can be computed in linear time with respect to the amount of data.

Another possibility is the following: we pick the value that has more information in common with all other values in the dataset than any other candidate. Unfortunately, this requires $O(n^2)$ time in the worst case. Still, this method is especially a reasonable thing to do if we assess classification performance with a similar measure of commonality. Therefore, we leave the description of this method to section 3.4 on assessing accuracy.

Back to our example and the default branch after choosing variable *drink*, we now have the following right hand side of data:

(Low, $High$).
(Low, Low).
(Low, $High$).
($High$, $High$).

No matter, which method of computing the most likely value we choose from above, the estimate for the most likely value will be (Low, $High$). Therefore, our model now looks as follows:

```
let
  (v1, v2) =
    case drink of
      Water → ...
      Beer → ...
      _ → (Low, High)
in
(Very v1, v2)
```

In the branches still missing for drinks *Water* and *Beer* we only need to start all over with the new partitions, normalizing and recursively partitioning as required. Finally, we would get the following model:

```
let
  (v1, v2) =
    case drink of
      Water → (Low, High)
      Beer →
       case meal of
         WienerSchnitzel _ → (Low, Low)
         TapirSoup spiced →
           let
             v3 =
               case spiced of
                 False → High
                 True → Low
           in
           (v3, High)
         _ → (Low, High)
      _ → (Low, High)
in
(Very v1, v2)
```

The big advantage of such models is, as we can see, that the *let-abstractions* allow us to predict parts of structured values before we have knowledge (due to discrimination) about others. It might be even more human readable if we used *where-abstractions*, which place the abstracted part after the one where it is to be inserted, but this is only a minor syntactic issue.

3.4. Assessing accuracy

We first define *information contents* for terms: it expresses the number of bits required to encode a given term in a given set of type equations. If, for example, some value is of a sum type that consists of four constructors, we would need two bits to encode the topmost constructor.

Then we continue with all substructures possibly associated with this constructor, counting bits as we go along. Given the type equations of our example, the information contents of *Pizza* (*Cheese Mozzarella*) would be $\log_2(3) + \log_2(3) + \log_2(2) = 4.17$ bits.

When comparing two values, we can now know how much information the two terms have in common by counting the number of bits of all constructors on which they agree. This gives us a numeric estimate of commonality. For instance, comparing the value from above to *Pizza* (*Cheese Gorgonzola*), we would get a common information contents of 3.17 bits (accuracy of 76%), because the two topmost constructors are the same. This way we can evaluate a model of the generalized decision tree learner on a test set and compare the results to the expected values, which allows us to compute a percentage of correctly predicted bits.

By computing the average common information contents for all values in a set, we can choose the one that has the highest amount of common bits with all other values, which is one possible method of calculating a most likely value.

3.5. Model factorization

Even though the algorithm can sometimes immediately see that parts of structures can be factored out using let-abstractions, in other cases this is only possible after sub-models have been learnt. In the standard decision tree learner C4.5 this transformation is coined *suing* and can there only be usefully applied to leaf nodes. E.g. consider this example data that maps a boolean to a satisfaction value:

$$
\begin{aligned}
&\mathit{True} \rightarrow \mathit{Low}.\\
&\mathit{True} \rightarrow \mathit{Low}.\\
&\mathit{True} \rightarrow \mathit{High}.\\
&\mathit{False} \rightarrow \mathit{Low}.\\
&\mathit{False} \rightarrow \mathit{Low}.\\
&\mathit{False} \rightarrow \mathit{High}.
\end{aligned}
$$

The right hand side is not unique. But after splitting according to the boolean and computing the most likely value for each branch, we get the following model:

$$
\begin{aligned}
&\textbf{case } \mathit{boolean} \textbf{ of}\\
&\quad \mathit{True} \rightarrow \mathit{Low}\\
&\quad \mathit{False} \rightarrow \mathit{Low}
\end{aligned}
$$

As we can see, no matter what boolean we observe, the most likely value will be *Low*. Therefore, the choice can be eliminated and the

model would simply reduce to the single leaf node *Low*. Generalizing this to algebraic datatypes, this is quite a bit trickier. Let's extend the above example to two satisfaction values on the right hand side of the data:

$$
\begin{array}{l}
\mathit{True} \rightarrow (\mathit{Low},\ \mathit{Low}).\\
\mathit{True} \rightarrow (\mathit{Low},\ \mathit{Low}).\\
\mathit{True} \rightarrow (\mathit{High},\ \mathit{Low}).\\
\mathit{False} \rightarrow (\mathit{Low},\ \mathit{High}).\\
\mathit{False} \rightarrow (\mathit{Low},\ \mathit{High}).\\
\mathit{False} \rightarrow (\mathit{High},\ \mathit{High}).
\end{array}
$$

This time the model for this data will look as follows:

$$
\begin{array}{l}
\textbf{case}\ \mathit{boolean}\ \textbf{of}\\
\quad \mathit{True} \rightarrow (\mathit{Low},\ \mathit{Low})\\
\quad \mathit{False} \rightarrow (\mathit{Low},\ \mathit{High})
\end{array}
$$

Now we cannot reduce this to a leaf node anymore. But we can factor out the left part of the pairs in each branch using a let-abstraction:

$$
\begin{array}{l}
\textbf{let}\\
\quad v1 =\\
\qquad \textbf{case}\ \mathit{boolean}\ \textbf{of}\\
\qquad\quad \mathit{True} \rightarrow \mathit{Low}\\
\qquad\quad \mathit{False} \rightarrow \mathit{High}\\
\textbf{in}\\
(\mathit{Low},\ v1)
\end{array}
$$

We would even have to consider common substructures and also expect cases where let-abstractions share common structure with leaf nodes. In extreme scenarios parts of a leaf node could even propagate through let-abstractions up to the top of the model! Though it is a bit challenging to implement this transformation, the algorithm itself is efficient and always yields a unique solution, i.e. a minimal factorization of a model[19]. The obvious advantage of such model rewritings is lower model complexity, which leads to more informative models for humans and allows more accurate model comparisons in terms of complexity for estimating the generalization capability.

[19] This, too, is implemented in the *AIFAD*-system.

3.6. Handling products of products

We have so far left out the point that it may be convenient for the user to specify data using products of products. This can make specifications more modular, e.g.:

$$\begin{aligned}&\textbf{data}\ \mathit{Kind} &&= \mathit{Starter} \mid \mathit{Main}\\ &\textbf{data}\ \mathit{Size} &&= \mathit{Small} \mid \mathit{Large}\\ &\textbf{type}\ \mathit{Meal} &&= (\mathit{Kind},\ \mathit{Size})\\ &\textbf{type}\ \mathit{Meals} &&= (\mathit{Meal},\ \mathit{Meal})\end{aligned}$$

A legal value of type *Meals* could be:

$$((\mathit{Starter},\ \mathit{Small}),\ (\mathit{Main},\ \mathit{Large}))$$

To treat such nested products efficiently within the algorithm, it is generally a good idea to flatten them internally, i.e. that each product consists of sums only. We thus invent an internal type for *Meals* that looks as follows:

$$(\mathit{Kind},\ \mathit{Size},\ \mathit{Kind},\ \mathit{Size})$$

and whose values are converted accordingly.

The same flattening happens with arguments of constructors. This way the algorithm can directly iterate over sums (e.g. during entropy calculation) instead of having to decompose products for each computation. Unflattening, which we have to do when converting internal values back to their original representation, is as easy a transformation as flattening, because the flattened and unflattened specifications are isomorphic.

3.7. Efficiency of generalized decision tree learning

The *AIFAD*-implementation of the generalized algorithm has the same time complexity as ordinary decision tree learning, including structured data. Comparative experiments with C4.5 show that there is usually only a factor of two to five[20] difference in speed, even though the implementation of the algorithm is not specialized towards unstructured representations, implemented in the high-level functional language *OCaml*[21] rather than the low-level language *C* and not yet fully optimized.

[20] A factor of five is usually only observed on data containing many missing values, which are encoded in a structured way.

[21] For details on this languages, see: http://www.ocaml.org

What concerns memory consumption, there is a small disadvantage: since structured data does not allow for certain in-place operations without significant loss of time efficiency, memory requirements are higher than for C4.5. More precisely, there is an additional logarithmic factor involved, which depends on the number of samples. This, however, has only lead to memory problems in our experiments once: on a dataset containing about 600,000 samples and more than 40 variables, which is huge even in industrial terms. It very rarely happens that *AIFAD* consumes more than ten times as much memory as C4.5, and logarithmic factors usually become negligible with high numbers of samples.

4. Practical aspects

We now consider practical aspects arising from the increased expressiveness provided by algebraic datatypes for the purpose of machine learning.

4.1. Novel solution to the missing value problem

In most machine learning systems missing values receive special treatment. Unfortunately, this often does not suit well to the particular application, because there may be different reasons why values are missing: they may have been dropped because they were not observable, they may not make sense in a specific context, etc. Therefore, a uniform treatment does not really solve the problem.

A straightforward and interesting solution to the treatment of missing values is provided by algebraic datatypes. For example, let's assume we have a variable of type t consisting of values A and B, but that also allows for missing values. Then we could encode this as follows:

$$\begin{aligned}&\textbf{data}\ t &&= A \mid B\\ &\textbf{data}\ \mathit{maybe\_t} &&= \mathit{Missing} \mid \mathit{Observed}\ t\end{aligned}$$

This way the machine learning system can explicitly learn models for cases where missing values occur without having to discriminate on t, which encodings like

$$\textbf{data}\ \mathit{maybe\_t} = A \mid B \mid \mathit{Missing}$$

would enforce.

More about recoding existing data specifications using algebraic datatypes, advantages that can be achieved this way and more details

on how to treat missing values in a fine-grained way can be found in (Mottl, 2002).

4.2. Expert systems

The lack of structure on both input and output values, as is the case for attribute-value representations, makes decision trees not very suitable for expert systems: some data may be collected in a step-wise manner, yielding ever increasing levels of detail. Since only structured attributes allow this, an expert system building on normal decision trees would sometimes force the user to reveal more information in one step to the system than necessary for particular queries. The dual case is that users may only want to know certain aspects of an output value, but normal decision trees would require them to provide sufficient information until a leaf node is reached, even though the partial answer might already be predictable at a much earlier point of time.

As we have explained in section 3.1.2, the presented algorithm (including model factorization) guarantees that a maximum of information is predicted with a minimum of input. Therefore, when a user enters data, the system can immediately tell, what can already be predicted from this information. Dually, when the user asks for a specific part of the result, the system will exactly ask questions only that are really required for determining the requested value.

4.3. Natural language processing

There is a tight relationship between algebraic datatypes and context-free grammars (CFGs)[22]: from a mathematical point of view they are both so-called *semirings*, i.e. they are both defined using equations involving a product operator (product types vs. string concatenation) and a sum operator (sum types vs. alternative productions), both of which are semigroup operations. Types correspond to nonterminals of a CFG while the creation of sum types (alternatives) corresponds to alternative productions of a certain nonterminal. Arguments of constructors correspond to the structure of a production.

Thus, there is always an isomorphic data specification for context-free grammars involving algebraic datatypes. For example, let's take the following CFG in Backus-Naur form (BNF):

[22] See e.g. (Floyd and Beigel, 1994) for an introduction to the syntax of formal languages.

```
<Sentence> := <Subject> <Predicate> | <Interjection> "!"
<Subject> := "He" | "She"
<Predicate> := "eats" | "sleeps"
<Interjection> := "Wow" | "Ouch"
```

A corresponding set of type definitions could be specified as follows:

data *sentence* = *SP subject predicate* | *IJ interjection*
data *subject* = *He* | *She*
data *predicate* = *Eats* | *Sleeps*
data *interjection* = *Wow* | *Ouch*

As we can see, each production requires the introduction of a new constructor. If some production contains structure, this constructor (e.g. *SP*) takes corresponding arguments for each nonterminal in the production. Recursive grammars imply recursive type equations.

Applying the generalized decision tree learner to two such encoded grammars (one for the source language, one for the target language) and sample data (parse trees of words in the two context-free languages), the system essentially induces tree transducers for derivation trees of context-free languages. The resulting algebraic datatypes can be unambiguously converted to words of the context-free target language again.

5. Theoretical aspects

We would like to point out here that there is a strong relation between type systems and logic. More precisely, the so-called *Curry-Howard correspondence*[23] relates types with propositions, and programs with proofs. It can be shown that type inference rules for the typed lambda calculus (i.e. functional programs) relate to natural deduction[24] in propositional logic. While this is traditionally presented for implication introduction and elimination (type inference rules for function abstraction and application) only, it is also possible to find such relations for algebraic datatypes. In the following derivations the type declarations (right side of colons) in the type inference rules are structurally the same as the inference rules of propositional logic presented next to them.

Construction of values of sum type (here: two constructors Inl and Inr, each taking an argument of different type) corresponds to introduction of disjunctions ($\vee$I):

[23] See e.g. (Stenlund, 1972) for details.
[24] A thorough introduction to natural deduction can be found in (Girard, 1987).

$$\frac{\begin{array}{c}\vdots\\ t\ :\ \tau_1\end{array}}{\mathrm{Inl}(t) : \tau_1|\tau_2} \qquad \frac{\begin{array}{c}\vdots\\ A\end{array}}{A \vee B} \qquad\qquad \frac{\begin{array}{c}\vdots\\ t\ :\ \tau_2\end{array}}{\mathrm{Inr}(t) : \tau_1|\tau_2} \qquad \frac{\begin{array}{c}\vdots\\ B\end{array}}{A \vee B}$$

Pattern matching on values of sum type corresponds to elimination of disjunctions ($\vee$E):

$$\frac{\begin{array}{ccccc} & & [\![x_1\ :\ \tau_1]\!] & & [\![x_2\ :\ \tau_2]\!] \\ \vdots & & \vdots & & \vdots \\ t\ :\ \tau_1|\tau_2 & & t\ :\ \tau & & t\ :\ \tau \end{array}}{\mathrm{case}\ t\ \mathrm{of}\ \mathrm{Inl}(x_1).t_1,\ \mathrm{Inr}(x_2).t_2 : \tau} \qquad \frac{\begin{array}{ccc} & [\![A]\!] & [\![B]\!] \\ \vdots & \vdots & \vdots \\ A\ \vee\ B & C & C \end{array}}{C}$$

Construction of values of product type (pairs) corresponds to introduction of conjunctions ($\wedge$I):

$$\frac{\begin{array}{cc}\vdots & \vdots \\ t_1\ :\ \tau_1 & t_2\ :\ \tau_2\end{array}}{(t_1, t_2) : (\tau_1, \tau_2)} \qquad \frac{\begin{array}{cc}\vdots & \vdots \\ A & B\end{array}}{A \wedge B}$$

And finally, decomposition of pairs corresponds to elimination of conjunctions ($\wedge$E):

$$\frac{\begin{array}{c}\vdots\\ t\ :\ (\tau_1, \tau_2)\end{array}}{\mathrm{fst}(t) : \tau_1} \qquad \frac{\begin{array}{c}\vdots\\ A \wedge B\end{array}}{A} \qquad\qquad \frac{\begin{array}{c}\vdots\\ t\ :\ (\tau_1, \tau_2)\end{array}}{\mathrm{snd}(t) : \tau_2} \qquad \frac{\begin{array}{c}\vdots\\ A \wedge B\end{array}}{B}$$

Considering this relation, the type of the data in attribute-value representations would correspond to propositions that are in *conjunctive normal form* (CNF), i.e. conjunctions of disjunctions, whereas algebraic datatypes relate to arbitrary propositions involving these logical connectives. Any formula in propositional logic can be translated to CNF, but as is well known, this may require exponential space (and hence time) in the worst case[25]. This, too, shows that our more general representation can yield exponential improvements over attribute-value encodings concerning size of unambiguous representation of structured data and induced models.

The demonstrated correspondence may henceforth be used to convert induced functional programs involving algebraic datatypes to proofs of logical propositions. Since logical propositions are especially relevant for complexity analysis within the PAC-framework of computational

[25] For details on such rewritings see e.g. (Bundy, 1983).

learning theory[26], this bridge between two fields of research in machine learning may turn out to be very fertilizing.

6. Future work

6.1. Polymorphic recipes

Further extensions to the presented data specifications might involve so-called *polymorphic types*. These are essentially types that accept type parameters. This allows us to greatly modularize data specifications. For example, instead of specifying missing values as in section 4.1, we could define a polymorphic type for them that looks as follows:

data *Optional Value* = *Missing* | *Observed Value*

Then we could quickly create new definitions allowing missing values for arbitrary other types, e.g. some type T, by just writing *Optional* T, thus greatly simplifying large, complex specifications that consist of similar data patterns.

6.2. Exotic meals made from abstract cooking techniques

So far we have avoided speaking about numerical values. Of course, adding support for those will be mandatory for many real-world applications. However, instead of just adding this special case, why not add support for inducing functions over *abstract datatypes*? By providing signatures of (possibly even many-sorted) algebras, whose objects may be but are not limited to numbers, and that define arbitrary operators on them (e.g. trigonometric functions, matrix operations, set operations, etc.), we could induce arbitrary programs. This would then not only cover classification tasks but also regression and many others.

6.3. Recursive functions

Efficiently inducing recursive functions, ones which can operate on recursively defined data, is a very challenging task. Most attempts have failed due to the pervasiveness of non-termination issues, which arise from general recursion. However, results from *category theory*[27] about certain higher-order functions (*fold* and *unfold*), which can be automatically derived from type definitions involving algebraic datatypes, seem

[26] See e.g. (Kearns and Vazirani, 1994) for an introduction.
[27] See e.g. (Taylor, 1999) for details on category theory.

promising[28]: they allow limiting the class of functions to be induced to certain recursion classes, e.g. the primitive recursive functions, which always terminate[29].

Acknowledgements

The author wishes to thank Gerhard Widmer for comments on an earlier draft.

This research is supported by the project "A New Modular Architecture for Data Mining (P12645-INF)", financed by the Austrian *Fonds zur Förderung der wissenschaftlichen Forschung (FWF)* of the Austrian Federal Ministry for Education, Science, and Culture. The Austrian Research Institute for Artificial Intelligence acknowledges basic financial support from the Austrian Federal Ministry for Education, Science, and Culture.

References

Bowers, A. F., C. Giraud-Carrier, and J. W. Lloyd: 2000, 'Classification of Individuals with Complex Structure'. In: *Proc. 17th International Conf. on Machine Learning*. pp. 81–88, Morgan Kaufmann, San Francisco, CA.

Bundy, A.: 1983, *The Computer Modelling of Mathematical Reasoning*. New York: Academic Press.

Flach, P. A.: 2000, 'The use of functional and logic languages in machine learning'. In: M. Alpuente (ed.): *Ninth International Workshop on Functional and Logic Programming (WFLP2000)*. pp. 225–237, Universidad Politecnica de Valencia. Invited talk.

Floyd, R. W. and R. Beigel: 1994, *The Language of Machines*. Freeman.

Girard, J.-Y.: 1987, *Proof Theory and Logical Complexity*, Vol. 1. Bibliopolis.

Kearns, M. J. and U. V. Vazirani: 1994, *An Introduction to Computational Learning Theory*. Cambridge, Massachusetts: The MIT Press.

Meijer, E. and G. Hutton: 1995, 'Bananas in space: extending fold and unfold to exponential types'. In: *Functional Programming & Computer Architecture*. San Diego, US.

Mitchell, T. M.: 1996, *Machine learning*. New York, US: McGraw Hill.

Mottl, M.: 2002, 'Modelling Large Datasets Using Algebraic Datatypes: A Case Study of the CONFMAN Database'. Technical Report 2002-27, Austrian Research Institute for Artificial Intelligence.

Quinlan, J. R.: 1986, 'Induction of decision trees'. *Machine Learning* **1**(1), 81–106. QUINLAN86.

[28] A good introduction to this approach of deriving fold/unfold-functions from type definitions can be found in (Meijer and Hutton, 1995).

[29] Or are *continuous* on infinite data. See (Winskel, 1993) on the relevance of continuity for describing the observable behavior of programs.

Quinlan, J. R.: 1992, *C4.5: Programs for Machine Learning.* Morgan Kaufmann.

Shannon, C. E. and W. Weaver: 1949, *A Mathematical Theory of Communication.* Urbana, Illinois: University of Illinois Press.
The classic introduction to Shannon's information theory, which generalizes Boltzman's measure of thermodynamic entropy S.

Stenlund, S.: 1972, *Combinators, Lambda-Terms and Proof Theory.* Dordrecht: D. Reidel.

Taylor, P.: 1999, *Practical Foundations of Mathematics.* Cambridge: Cambride University Press.

Winskel, G.: 1993, *The Formal Semantics of Programming Languages: An Introduction.*, Foundations of Computing series. MIT Press.