Inductive Inference 1.1

Lloyd Allison
School of Computer Science and Software Engineering,
Monash University,
Clayton, Victoria, Australia 3800.
http://www.csse.monash.edu.au/∼lloyd/tildeFP/II/

Technical Report 2004/153
(submitted 3 May, revised 13 May)

May 13, 2004

Abstract

Examined, the succinct expression of general solutions to inductive inference problems. Haskell types and type classes define the properties of various kinds of statistical model – distributions, function models and time-series. This is an application of Haskell which itself has applications, and is almost as general as Haskell's own area of application. Case studies in inductive inference, including mixtures of Markov models, state-based time-series, missing data, and mixed Bayesian networks, illustrate the functional style of programming with models. Polymorphic types, type inference, high-order functions and lazy evaluation are all useful.

Keywords: Bayesian networks, inductive inference, machine learning, minimum message length, MDL, MML, statistical models.

1 Introduction

Q: 'Function' is to 'functional programming' as 'statistical model' is to what?
A: Please send suggestions to the above address.

The report's aim is to express succinctly general solutions to inductive inference problems (Allison 2003a). The problems come from machine learning and data mining, and solutions take the form of *statistical models* and their estimators. Here there are two distinct ways in which we can be succinct. The first is in writing programs to solve new problems: a *general* system will tend to produce reusable solutions, an *expressive* system will make it easy to write new solutions that are succinct, and general. The second comes from a preference to return a "simple" model rather than a "complex" one, unless the latter is really necessitated by the data, i.e. avoid over-fitting.

The functional programming language Haskell is chosen (Allison 2004) because it is expressive and because it has a powerful system of polymorphic types and type classes. Functional programming encourages the composition of functions, and polymorphic types lead to general solutions. When it comes to inductive inference it is statistical models and functions on them that are to be composed.

The problem of over-fitting is well known. William of Occam argued, in medieval times, that explanations should be kept simple unless necessity dictates otherwise. A computer program doing inductive inference must address model complexity in some way. If models are to be composed to make new models the complexity of the composition must be dealt with. It is argued that the minimum message length (MML) criterion (Wallace and Boulton 1968) is a natural partner for functional programming in this domain.

Useful statistical models, and estimators, have been defined and can be used as building blocks in more complex models; using a good type and type-class system reveals their true generality (Allison 2003a). To quote Peyton Jones et al (2000, p280) on a different domain – the domain of the 'compositional denotational semantics' of financial contracts – 'At this point, any red blooded functional programmer should start to foam at the mouth, yelling "build a combinator library"', a vivid image! Such libraries have been defined for various domains, e.g. parsing, graphics, etc. (van Deursen et al 2000). In many of those domains the data types involved are quite specific, e.g. parsing is concerned with characters, symbols (strings) and trees. But in the present case, statistical models, operators and data can be very general – pretty much any computable model inferred from almost any type of data by an arbitrary algorithm. That is almost as general as the "application domain" of Haskell itself. And a 'compositional denotational semantics' of statistical models needs a compositional solution to the over-fitting problem; MML is advocated.

Inductive inference is an application of Haskell, but it is an application that is general and itself has many applications. This report does not "just" describe a Haskell program but it does use pieces of Haskell programs from case studies to illustrate programming with statistical models.

The report covers code version 200312 which updates 200309 (Allison 2003b). All code was compiled under the Glasgow Haskell Compiler, ghc, version 6.0.1. Local readers can find the case studies of
mixtures of Markov models in .../II/200312/20031218/,
Bayesian networks in .../II/200312/20031229/ and
stateful time-series in .../II/200312/20040311/.

2 MML

Minimum message length (MML) inference (Wallace and Boulton 1968, Wallace and Freeman 1987) builds on Shannon's 'mathematical theory of communication' (1948), hence 'message', and on Bayes (1763):

```
Pr(M&D) = Pr(M).Pr(D|M) = Pr(D).Pr(M|D)
```

```
msgLen(E) = -log(Pr(E))
msgLen(M&D) = msgLen(M)+msgLen(D|M) = msgLen(D)+msgLen(M|D)
```

where M is a model (theory, hypothesis, parameter estimate) of prior probability Pr(M) over some data, D, and E is an event of probability Pr(E), and msgLen(E) is the length of a message, in an optimal code, announcing E.

MML considers a *transmitter* sending a two-part message to a *receiver*. The first part, of length msgLen(M), states a model which is an answer to some inference problem. The second part, msgLen(D|M), states the data encoded as if the answer, M, is true; the receiver cannot decode the second part without the first part. If the space of models is finite or even enumerable each plausible model can be given a non-zero probability, but if one or more continuous parameters are involved this cannot be done unless the parameters are stated to *finite*, optimal accuracy. Strict MML (SMML) is concerned with the design of such optimal code books. Unfortunately SMML is infeasible for most inference problems (Farr and Wallace 2003). Fortunately there are efficient, accurate MML approximations (Wallace and Freeman 1987) for many useful problems and models.

MML is a *compositional* criterion because the complexity of data, models and sub-models are all measured in the same units: "[It is possible] to use [message] length to select among competing sub-theories at some low level of abstraction, which in turn can form the basis (i.e., the 'data') for theories at a higher level of abstraction. There is no guarantee that such an approach will lead to the best global theory, but it is reasonable to expect in most natural domains that the resulting global theory will at least be near-optimal" (Georgeff and Wallace 1984). In other words MML is a good fit with functional programming for inductive inference. MML has been used to assess the complexity of combined models of some specific types (e.g. Georgeff and Wallace 1984, Allison et al 1999) but its full *programming* potential has only recently started to be studied (Allison 2003a), hence the question at the start of the introduction. A functional language with a parametric polymorphic type system is a sound foundation for such developments.

3 Statistical Models

Haskell classes were previously defined (Allison 2003a) for models, function models (regressions) and time-series models. A model can return the probability, `pr`, and the negative log probability, `nlPr`, of a datum from its data-space. It can also compute the second-part, `msg2`, and the total message length, `msg`, for a data set. A function model returns a model of its output space conditional on a value from its input space. A time-series returns models for the elements of a sequences (list), each one possibly conditional on the previous values. A super-class, `SuperModel`, states that they must all returns their own prior probability, their own message length, `msg1`, and that each must be able to form a mixture of instances of itself. They must also be in class `Show` so that we can see the answers to inference problems.

```
class (Show sMdl) => SuperModel sMdl where
 prior    ::sMdl -> Probability
 msg1     ::sMdl -> MessageLength
 msg1Base ::Double -> sMdl -> MessageLength
 mixture  ::(Mixture mx, SuperModel (mx sMdl)) => mx sMdl -> sMdl

class Model mdl where
 pr   ::(mdl dataSpace)->dataSpace->Probability
 nlPr ::(mdl dataSpace)->dataSpace->MessageLength
 nlPrBase ::Double->(mdl dataSpace)->dataSpace->MessageLength
 msg  ::SuperModel (mdl dataSpace) =>
        (mdl dataSpace)->[dataSpace]->MessageLength
 msgBase ::SuperModel (mdl dataSpace) =>
           Double->(mdl dataSpace)->[dataSpace]->MessageLength
 msg2 ::(mdl dataSpace)->[dataSpace]->MessageLength
 msg2Base ::Double->(mdl dataSpace)->[dataSpace]->MessageLength

class FunctionModel fm where
 condModel ::(fm inSpace opSpace)->inSpace->ModelType opSpace
 condPr    ::(fm inSpace opSpace)->inSpace->opSpace->Probability
 condNlPr  ::(fm inSpace opSpace)->inSpace->opSpace->MessageLength
 condNlPrBase ::Double ->
             (fm inSpace opSpace)->inSpace->opSpace->MessageLength

class TimeSeries tsm where
 predictors ::(tsm dataSpace)->[dataSpace]->[ModelType dataSpace]
 prs        ::(tsm dataSpace)->[dataSpace]->[Probability]
 nlPrs      ::(tsm dataSpace)->[dataSpace]->[MessageLength]
 nlPrsBase  ::Double ->
              (tsm dataSpace)->[dataSpace]->[MessageLength]
```

Useful statistical models, including multi-state, normal and multivariate distributions, mixture models, Markov models, finite function-models (conditional probability tables) and classification trees, have been defined and made instances of the appropriate classes (Allison 2003b). Here, these building blocks are extended, tested and applied to new inductive inference problems to explore and illustrate the style of programming.

4 Case Study: Mixtures of Markov Models

The problem of modelling mixtures of Markov models was posed in a tea-room conversation: Given a population of sequences over a finite (`Bounded`, `Enum`) alphabet (type), is the population a mixture of $k \geq 1$ sub-populations, each sub-population being described by its own Markov model? This is unsupervised classification (clustering) of sequences. An expectation maximization (EM) algorithm, `estMixture`, had already been defined (Allison 2003a) for mixture modelling, the "intended" application being over multi-variate data.

Function `estMixture` requires estimator(s) for the components of the mixture. Such an estimator must deal with *weighted* data because fractional assignment of data to components is needed to give unbiased estimates. The existing estimator for Markov models of order-k did not handle weighted data so, for a proof of concept, a suitable estimator for first-order Markov models was created. Given a weighted data set (list) of several sequences, each sequence is scanned with a copy of itself shifted one position so as to align the previous and current elements. This gives training pairs from which to estimate a finite function model, `ff`, which is the basis of a finite list function model, `flf`. The latter is turned into a time-series model, and thence into a model (which includes consideration of sequence length) of sequences, by standard conversion functions:

```
est_MM1_wtd trainingSeqs weights =  -- est 1st order Markov Model
 let scanOneSeq xs w rest =
      let scan (i:is) (o:os) = (i,o,w) : (scan is os)
          scan   _      _    = rest
      in scan xs (tail xs)
     scanManySeqs (s:ss) (w:ws) = scanOneSeq s w (scanManySeqs ss ws)
     scanManySeqs  _      _      = []   -- done
     (ips,ops,ws) = unzip3(scanManySeqs trainingSeqs weights)
     ff =  estFiniteFunctionWeighted ips ops ws
     predictor (x:_) = condModel ff x
     predictor  []   = uniform (elt minBound) (elt maxBound)
     elt x = x `asTypeOf` (trainingSeqs!!0!!0)
     flf = FM (msg1 ff) predictor (\() ->show ff)
 in (timeSeries2model.functionModel2timeSeries) flf

e.g. mix2 = estMixture [est_MM1_wtd, est_MM1_wtd] trainSet
```

The exercise suggests that every time-series estimator should probably work from a set (list) of sequences rather than a single sequence; it is in any case easy enough to one thing into a set of things but not vice-versa.

5 Case Study: Stateful Time-Series

As previously defined (Allison 2003a), a time-series model was based on a function from the *context* of past values to a model of the next element. It was well known and quite clear that time-series models could also be defined from *stateful* functions and experiments have now been carried out.

```
data TimeSeriesType' = ... |
 forall state.
  TSMs MessageLength
```

```
      state                              -- initial state
      (state -> dataSpace -> state)      -- state-transition fn
      (state -> ModelType dataSpace)     -- predict/model next elt'
      (() -> String)                     -- Show it
```

The new option is made an instance of class `TimeSeries` in the obvious way. The type of the state must be hidden as an existential type (`forall`!) within `TimeSeriesType'`; existential types are available in ghc as a type extension to Haskell-98. If not hidden, the state must be a type parameter to `TimeSeriesType'` and that would make every instantiation a different type, e.g. putting an end to mixtures of arbitrary time-series models over a given data-space.

A parameterless time-series model based on the idea of an *adaptive* code provides an example. It assumes only that the data are homogeneous and come from an unknown zero-order source. Counters are kept for the number of occurrences of each possible value from the start of the sequence up to the current position. The counters are initialised to one and, together with the total (for convenience), form the initial state. The state transition function increments the appropriate counter, and the total. The model at a given position is derived by turning the frequencies into probability estimates.

```
adaptive =
 let mn = minBound
     size = (fromEnum (maxBound `asTypeOf` mn)) - (fromEnum mn)+1
     state0 = (size, replicate size 1)     -- initial state
     t (total, counts) datum =             -- state transition fn
       (total+1,
        increment (fromEnum datum -
          fromEnum (mn `asTypeOf` datum)) counts)
     p (total, counts) =                   -- :: state -> model
       ((modelInt2model mn) . probs2model
       . (map (/ (fromIntegral total)))) counts
 in TSMs 0 state0 t p (\()->"adaptive")
```

Incidentally, Wallace and Boulton (1968) showed that calculations of the information content of a sequence of data, from such a source, under the uninformative adaptive and "combinatorial" codes give identical results, as one would hope. And the method which first states an estimate of the probabilities to *optimal*, finite precision gives a total message length which is greater by a fraction of a bit per parameter, that being the small price of transmitting an opinion.

6 Case Study: Lost Person

Koester's (2001) lost-person data set has been studied in CSSE, Monash (Twardy 2002, + Hope 2004). There are 363 records, and 15 attributes, numbered 0-14, but attention has mostly been restricted to the first eight attributes. One hope

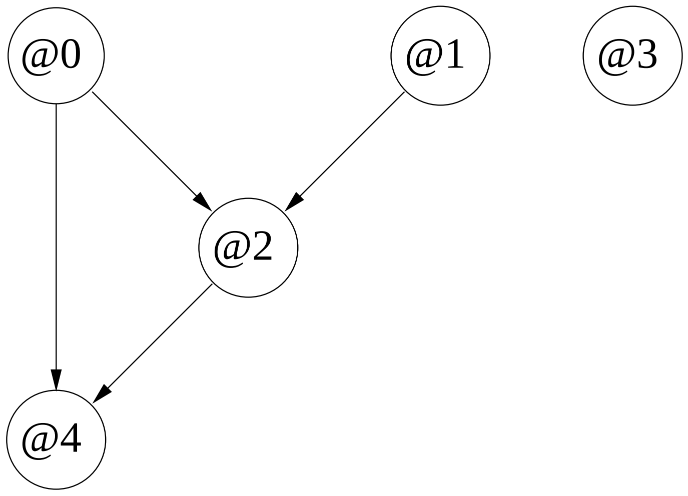

Figure 1: A Simple Example Network.

```
Net:[
{CTleaf N(1.0,0.41)(+-0.1),_,_,_,_},          -- @0 ~ N(1,0.4)

{CTleaf _,mState[0.5,0.5],_,_,_},             -- @1

{CTfork @0<|>=1.4[                            -- @2 | @0,@1
  {CTleaf _,_,mState[0.99,0.01],_,_},           -- @0<1.4
  {CTfork @1=False|True[                        -- @0>=1.4
    {CTleaf _,_,mState[0.98,0.02],_,_},           -- @1 = False
    {CTleaf _,_,mState[0.02,0.98],_,_}]}]},       -- @1 = True

{CTleaf _,_,_,mState[0.5,0.5],_},             -- @3, independent

{CTfork @2=False|True[                        -- @4 | @0, @2
  {CTfork @0<|>=1.0[                          -- @2=False
    {CTleaf _,_,_,_,N(0.55,0.2)(+-0.1)},        -- @0 < 1.0
    {CTfork @0<|>=1.4[                          -- @0 >= 1.0
      {CTleaf _,_,_,_,N(1.0,0.2)(+-0.1)},         -- @0 [1.0,1.4)
      {CTleaf _,_,_,_,N(1.45,0.2)(+-0.1)}]}]},    -- @0 >= 1.4
  {CTleaf _,_,_,_,N(3.45,0.2)(+-0.1)}]}       -- @2=True
]
```

Figure 2: Trees in the Nodes of the Simple Example Network.

is to be able to predict distance travelled, `DistIPP` attribute 7, from attributes zero to six.

A Bayesian network is an appropriate tool to investigate relationships amongst the various attributes. Friedman and Goldszmidt (1996) suggested using decision-trees, i.e. classification trees, in place of the usual conditional probability tables (CPTs) within the nodes of such networks. A classification tree can "become" a full CPT in the limit but can be much more economical, that is less complex, in many cases. We just happen to have a general MML classification tree, in Haskell, and it can also test continuous attributes, and can have discrete or continuous distributions, or even regressions, in the leaves (Allison 2003a). Comley and Dowe (2003) have also used trees within the nodes of networks.

Figure 1 shows an example of a simple network inferred for an artificial data set having five attributes. A node represents an attribute. An edge represents a (direct) conditional dependence of a "child" on a "parent" and, in a suitable context, relates to causality. In the example, attribute 2 is conditionally dependent on attributes 0 and 1, attribute 3 is independent of the other attributes, and so on. Figure 2 shows the classification trees for all the network nodes. Each tree consists of fork-nodes, `CTfork`, and/or leaf-nodes, `CTleaf`, which are not to be confused with the network's nodes. A fork tests a parent attribute value and a leaf models the appropriate child attribute. A test on a discrete attribute is shown as e.g. `@2=False|True`, etc.. A test on a continuous attribute is shown as e.g.`@0<|>=1.4`, i.e. less than or greater than or equal to 1.4, etc.. The multi-state distribution, `mState`, is used to model a discrete attribute, and the normal distribution, N(m,s)(a), of mean m, standard deviation s and data measurement accuracy 'a', is used to model a continuous attribute.

The remainder of this section describes how Haskell can be used to investigate the lost-person data. The application serves as an example to illustrate the composition of statistical models: multi-state and normal distributions within classification trees within a Bayesian network.

6.1 Data

The first step is to define the attribute types in the (slightly simplified) data set:

```
data Tipe = Alzheimers| Child| Despondent|
            Hiker| Other| Retarded| Psychotic
  deriving (Eq, Enum, Read, Show, Bounded)
type Age  = Double
data Race = White | Black
  deriving (Eq, Enum, Read, Show, Bounded)
data Gender  = Male | Female
  deriving (Eq, Enum, Read, Show, Bounded)
data Topography = Mountains | Piedmont | Tidewater
  deriving (Eq, Ord, Enum, Read, Show, Bounded)
data Urban = Rural | Suburban | Urban
```

```
  deriving (Eq, Ord, Enum, Read, Show, Bounded)
type HrsNt   = Double  -- hours notified
type DistIPP = Double  -- distance
 ...
```

Haskell's standard Prelude (Peyton Jones et al 1999) instantiates tuples, up to 7-tuples, in classes `Read` and `Show`, so the 15-tuples here need to be made instances of those classes for input and output respectively. This is an easy, if tedious, job and could in principle be automated in template Haskell (Sheard and Peyton Jones 2002), say.

The model needs to *split*, i.e. partition, the data on various attribute values (Allison 2003b, 2004) for the benefit of the classification trees.

```
class Splits t where
  splits :: [t] -> [Splitter t]

data Splitter t = Splitter Int   (t -> Int) (() ->String)
                  -- i.e. arity partition_fn description
```

A continuous, ordered (`Ord`), attribute, such as `Age`, is split on being $<$ or $\geq$ some value. A discrete, i.e. `Bounded`, enumerated (`Enum`), attribute, such as `Gender`, of a k-valued type is usually split into k subsets, as defined by `splitsBE`. However `Topography` and `Urban` are `Bounded`, enumerated *and* ordered (`Ord`), so we also have the options of splitting each into two subsets on the basis of order, as defined by `splitsOrd`:

```
instance Splits Tipe       where splits = splitsBE
instance Splits Race       where splits = splitsBE
instance Splits Gender     where splits = splitsBE
instance Splits Topography where splits = splitsBE --or splitsOrd
instance Splits Urban      where splits = splitsBE --or splitsOrd
 ...
```

The question of which distribution, and therefore which estimator, to use for each attribute now arises. The standard estimator for the normal (Gaussian) distribution uses a uniform prior on the mean and an inverse prior on the standard deviation and requires their ranges, and also the data measurement accuracy:

```
e0 = (estModelMaybe estMultiState)                  -- Tipe
e1 = (estModelMaybe (estNormal 0 90 1 70 0.5))      -- Age
e2 = (estModelMaybe estMultiState)                  -- Race
e3 = (estModelMaybe estMultiState)                  -- Gender
e4 = (estModelMaybe estMultiState)                  -- Topography
e5 = (estModelMaybe estMultiState)                  -- Urban
e6 = (estModelMaybe (estNormal 0 200 1 100 0.5))
e7 = (estModelMaybe (estNormal 0 50 0.5 30 0.2))    -- DistIPP
 ...
```

Finally the individual estimators are assembled into a composite that matches a data tuple; all 15 component distributions are defined by the estimator but lazy evaluation ensures that only the selected eight, say, are evaluated.

```
estimator = estVariate15 e0 e1 e2 e3 e4 e5 e6 e7
                         e8 e9 e10 e11 e12 e13 e14
```

Function `estVariate15` estimates a 15-variate probability distribution which is an instance of class `Project` – a standard piece of inductive inference machinery. An instance, t, of `Project` is some multi-dimensional type for which a list of Boolean flags can be used to restrict t to certain selected dimensions. The non-selected dimensions must behave in a trivial, "identity" manner that is appropriate to type t. In the case of a `Model` this is to return zero message length, probability one, for non-selected attributes.

```
class Project t where
  select :: [Bool] -> t -> t
  selAll :: t -> [Bool]  -- all True flags
```

A class `Splits` was previously defined for partitioning data – discrete, continuous or multi-variate. A class `Splits2`, inspired by `Project`, is now defined (it could equally well be folded into `Splits`) to allow splitting on selected attributes:

```
class Splits2 t where
  splitSelect :: [Bool] -> [t] -> [Splitter t]
```

The situation for adding k-ary types to class `Project`, or k-tuples to `Splits2`, is similar to that for k-tuples with respect to classes `Read` and `Show`.

6.2 Missing Data

The lost-person data set is "difficult" because it contains, or perhaps it is better to say does not contain, many missing values. Many data have at least one missing value, and some have several. Every attribute is missing in some datum. Haskell has the ideal type to represent possibly missing values: `Maybe`. And high-order functions are used to define succinct operators to extend arbitrary statistical models to cover possibly missing values.

The function `modelMaybe` is a good example of a high-order function on models. It turns an *arbitrary* model, m2, of non-missing data, t, into the corresponding model of `Maybe t` where the value may be missing. It requires a model, m1, of `Bool` of whether the data is present (True) or missing (False).

```
modelMaybe m1 m2 =
 let negLogPr (Just x) = (nlPr m1 True) + (nlPr m2 x)
     negLogPr Nothing  =  nlPr m1 False
 in MnlPr (msg1 m1 + msg1 m2) negLogPr
          ... show method omitted ...
```

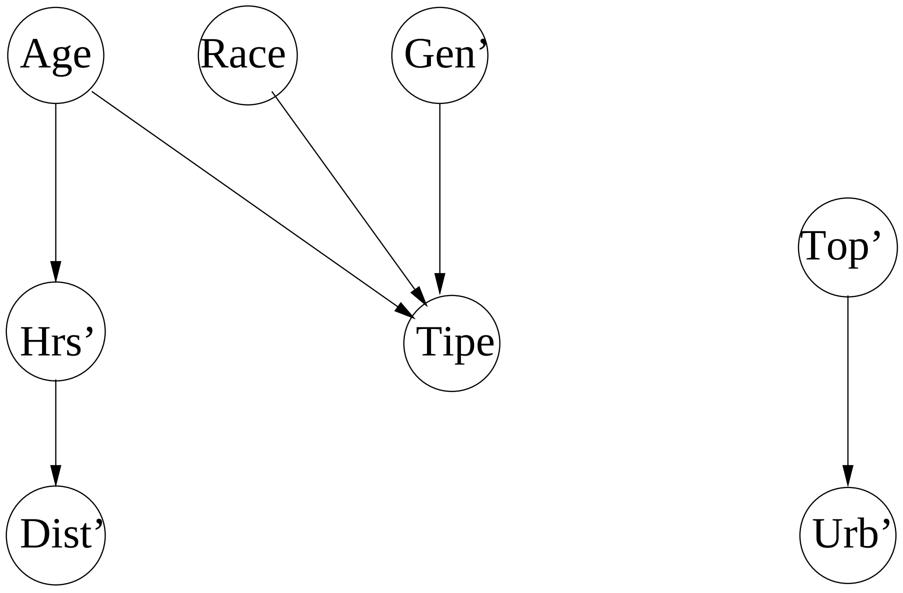

Figure 3: Lost Person Network.

There is a related estimator, `estModelMaybe`, which turns an estimator of non-missing data into the corresponding estimator where the data may include missing values:

```
estModelMaybe estModel dataSet =
 let present (Just _) = True
     present Nothing  = False
     m1 = uniformModelOfBool
     m2 = estModel (map (\(Just x) -> x) (filter present dataSet))
 in modelMaybe m1 m2
```

In the present application the missing-ness of values is certainly non-random for some attributes, for example `Age` is often omitted for cases of `Hiker::Tipe`. However, we are not interested in modelling missing-ness so a fixed 50:50 model, m1, is used above to "predict" missing (`Nothing`) or present (`Just`...). To estimate missing-ness, the following definition is used:

```
     m1 = estMultiState (map present dataSet)
```

Missing values also affect splits, i.e. partitions of the data. A simple strategy is for the attribute to be split as for the underlying type but with an extra option for `Nothing` cases:

```
maybeSplitter (Splitter n f d) =
 let f' Nothing  = n
```

```
    f' (Just x) = f x             -- Just x, as x was, 0..n-1
 in Splitter (n+1) f' (\() -> "("++d()++"|?)")  -- NB arity n+1
```

More complex strategies, not examined here, could try to predict in various ways what the missing value, or its distribution, really is and act on that.

6.3 The Mixed Bayesian Network Model

The function, `estNetwork`, for inferring a network needs to be given a permutation, a total ordering, of those attributes that are to be considered; it would be straightforward to search over permutations, heuristically if there were many attributes, but the simple algorithm does not do this and the permutation is taken to be common knowledge.

```
estNetwork  perm  estMV  dataSet =
 let
  n = (length . selAll) (estMV [])
  search _  []     = []  -- done
  search ps (c:cs) =     -- parents, ps, predict children, c:cs
   let
     opFlag  = ints2flags [c] n  -- identify the child ...
     ipFlags = ints2flags ps  n  -- ... and the parents
     cTree = estCTree (estAndSelect estMV opFlag)  -- leaf est'
                      (splitSelect ipFlags)        -- tests
                      dataSet dataSet              -- !
   in
     cTree : (search (c:ps) cs)     -- i.e. a list of trees
  trees     = search [] perm
  msgLen    = sum (map msg1 trees)  -- total complexity
  nlP datum = sum (map (\t -> condNlPr t datum datum) trees)

 in MnlPr msgLen nlP (\() -> "Net:" ++ (show trees))  -- Net'
```

The algorithm above uses the estimator for classification trees (Allison 2003a), estCTree, to do most of the work. The remainder consists of organising selector flags corresponding to allowed parents for the current node's child. Note that the `dataSet` seems to be passed to `estCTree` twice – as both input and output attributes, but auxiliary functions `ints2flags`, `estAndSelect` and `splitSelect`, use the flags to cause the child to be predicted (output) by the estimator and the parents to be used for splitting (input) as appropriate at each network node.

Cause and effect dictate that `Age`, `Race` and `Gender` cannot, in a causal sense, depend on other attributes and should come first, in some order, say [1,2,3]. `Tipe` probably depends on them, e.g. there are few young Alzheimers cases. `Topography` and `Urban` can sensibly come next, and one expects a relationship between them. That leaves `HrsNt` and finally `DistIPP` to make up a plausible ordering, [1,2,3,0,4,5,6,7]. There is also a natural null hypothesis which models the attributes independently:

```
dataSet = read (readFile dataFile) :: [MissingPerson]
nw = estNetwork [1,2,3,0,4,5,6,7] estimator dataSet
nullModel = estimator dataSet
```

Figure 3 shows the network inferred with an abridged node listing in figure 4. `Tipe`, attribute 0, is dependent on `Age`, `Gender` and `Race`. As expected, `Urban` is dependent on `Topography`. There is some direct dependence of `DistIPP` on `HrsNt`, and of the latter on `Age`, but there seems to be no strong predictor of `DistIPP` from other attributes. The model is significant, with a total two-part message length of 5512 nits against 5936 nits for the null model. Other analyses were tried, for example using ordered (`Ord`) splits on `Topography` and `Urban`, in place of `Bounded Enum` splits; the conclusions were much the same.

7 Conclusions

Functional programming's compositional style and Haskell's features have a number of advantages in inductive inference. Mapping a data set, such as lost-persons, onto the Haskell type system is a useful exercise in getting to know the data, very precisely. The necessity to define an attribute's properties, e.g. `Ord` or not, automatically suggests what is possible, e.g. to split `Topography` as discrete or as ordered (6.1). These things cannot be forgotten; the type and class system insists on bringing them to your attention.

In-built support for wide tuples, `(,)`, would make it easier to deal with typical multi-variate data sets, although template Haskell is a possible solution.

High-order functions, such as `modelMaybe` (6.2), are invaluable in creating new ways of using *arbitrary* models. The polymorphic type system ensures that the uses are general and remain type safe.

It is a common experience that polymorphic types often show a function to be more general than its programmer realised, e.g. the EM algorithm (4) applies to sequences not just to multivariate data.

Lazy evaluation means, for example, that only models of the selected attributes of lost-persons (6.1) are evaluated. Selections are made once at the top level, most of the algorithms do not "consider" the matter at all.

Computing model complexity by minimum message length (MML,2) is a good match with the compositional style of functional programming. The reader may hardly have noticed any explicit message length calculations. They are there in `modelMaybe`, for example, and are being passed around in the complexity of the network (6,6.3) and its classification trees (fig. 4) to direct the search.

A specific model can be created quickly to suit a new problem (4,5,6) thanks to Haskell's expressive power. Of course it cannot be claimed that the types and type classes for statistical models are of the best possible designs, e.g. a *case* can be made for specifying the notion of a *data set*; perhaps data traversal, data measurement accuracy and data weights should be wrapped up in suitable types and classes. Only experience, and time, will let us settle on the "best"

```
Net:[
@1, Age:
{CTleaf _,(Maybe 50:50,N(40.6,27.5)(+-0.5)),...},

@2, Race:
{CTleaf _,_,(Maybe 50:50,mState[0.66,0.34]),...},

@3, Gender:
{CTleaf _,_,_,(Maybe 50:50,mState[0.72,0.28]),...},

@0, Tipe:
{CTfork @1(<|>=19.0|?)[ ...uses @1, @2, @3... ]},

@4, Topography:
{CTleaf _,_,_,_,(Maybe 50:50,mState[0.17,0.52,0.31]),...},

@5, Urban:
{CTfork @4(=Mountains..Tidewater|?)[
  {CTleaf _,_,_,_,_,(Maybe 50:50,mState[0.93,0.04,0.04]),...},
  {CTleaf _,_,_,_,_,(Maybe 50:50,mState[0.70,0.19,0.11]),...},
  {CTleaf _,_,_,_,_,(Maybe 50:50,mState[0.38,0.02,0.6 ]),...},
  {CTleaf _,_,_,_,_,(Maybe 50:50,mState[0.73,0.2 ,0.07]),...}]},

@6, HrsNt:
{CTfork @1(<|>=62.0|?)[
  {CTleaf _,_,_,_,_,_,(Maybe 50:50,N( 8.7, 7.6)(+-0.5)),...},
  {CTleaf _,_,_,_,_,_,(Maybe 50:50,N(21.4,26.3)(+-0.5)),...},
  {CTleaf _,_,_,_,_,_,(Maybe 50:50,N(20.0,...1-case...),...}]},

@7, DistIPP:
{CTfork @6(<|>=1.0|?)[
  {CTleaf _,_,_,_,_,_,_,(Maybe 50:50,N( ...no-cases... ),...},
  {CTleaf _,_,_,_,_,_,_,(Maybe 50:50,N(0.59,0.6)(+-0.2)),...},
  {CTleaf _,_,_,_,_,_,_,(Maybe 50:50,N(1.52,2.8)(+-0.2)),...}]}]

network: 115.1 nits, data: 5396.6 nits
null:   5935.6 nits (@0..@7)
```

Figure 4: Trees in the Nodes of the Lost Person Network.

trade-off between generality, usability and efficiency. The lost-person case study took one and a half days to create, *including* how to handle missing data which had previously been in the "must think about that one day" category.

References

[1] Allison, L. (2003a) Types and classes of machine learning and data mining. 26th Australasian Computer Science Conference (ACSC), Adelaide, pp. 207–215, Feb 2003.

[2] Allison, L. (2003b) Inductive inference 1. *TR 2003/148*, School of Computer Science and Software Engineering, Monash University.

[3] Allison, L. (2004) Models for machine learning and data mining in functional programming. *J. Functional Programming*, to appear.

[4] Allison, L.; Powell, D. and Dix, T. I. (1999) Compression and approximate matching. *BCS Comput J.*, 42 (1), pp. 1–10.

[5] Bayes, T. (1763) An essay towards solving a problem in the doctrine of chances. *Phil. Trans. of the Royal Soc. of London*, 53, pp. 370–418. Reprinted in *Biometrika* 45(3/4), pp. 296–315, 1958.

[6] Comley, J. and Dowe, D. L. (2003) General Bayesian networks and asymmetric languages. Hawaii Int. Conf. Statistics and Related Fields (HICS-2), June 2003.

[7] Farr, G. E. and Wallace, C. S. (2002) The complexity of strict minimum message length inference. *BCS Comput. J.*, 45 (3), pp. 285–292.

[8] Friedman, N. and Goldszmidt, M. (1996) Learning Bayesian networks with local structure. UAI'96, pp. 252–262.

[9] Georgeff, M. P. and Wallace C. S. (1984) A general selection criterion for inductive inference. European Conf. on Artificial Intelligence (ECAI84), Pisa, pp. 473–482, September 1984. A longer version is available as Wallace, C. S. and Georgeff, M. P. (1983) A general objective for inductive inference. *TR 32*, Department of Computer Science, Monash University.

[10] Koester, R. J. (2001) Virginia dataset on lost-person behaviour. Author's site http://www.dbs-sar.com/.

[11] Peyton Jones, S. et al (1999) Report on the Programming Language Haskell-98. http://www.haskell.org/

[12] Peyton Jones, S.; Eber, J.-M. and Seward J. (2000) Composing contracts: an adventure in financial engineering. Proc. 5th Int. Conf. on Functional Programming, Montreal, pp. 280–292.

[13] Shannon, C. E. (1948) A mathematical theory of communication, *Bell Syst. Technical Jrnl.*, 27: pp. 379–423 and pp. 623–656.

[14] Sheard, T. and Peyton-Jones, S. (2002) Template meta-programming for Haskell. Proc. of the workshop on Haskell, ACM, pp. 1–16.

[15] Twardy, C. R. (2002) SAR*bayes*: Predicting lost person behavior. Presented to the National Association of Search and Rescue (NASAR 2002), Charlotte, NC. http://sarbayes.org/nasar.pdf

[16] Twardy, C. R. and Hope, L. R. (2004) Missing data on missing persons. Submitted.

[17] Van Deursen, A.; Lint P. and Visser, J. (2000) Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35 (6), pp. 26–36.

[18] Wallace, C. S. and Boulton, D. M. (1968) An information measure for classification. *BCS Comput. J.*, 11 (2), pp. 185–194.

[19] Wallace, C. S. and Freeman, P. R. (1987) Estimation and inference by compact coding. *J. Royal Statistical Society series B.*, 49 (3), pp. 240–265.