Designing and Building a Graphical Model Library in Standard ML

Clint Morgan

December 2004

Abstract

This paper discusses several design considerations for a probabilistic graphical model library. The library, dubbed GM, currently only contains a small subset of desirable features for such a library. In particular, it implements several techniques for performing inference in discrete graphical models.

Rather than introducing a complete library, this work discusses the library design and issues which arouse during implementation. Particular attention is paid to the benefits and burdens of developing such a library in Standard ML using the SML module system to structure the design.

1 Introduction

Probabilistic graphical models (PGMs) are used to express independence relations of random variables. There are many types of models that fall into the broad category of "graphical models". Network topology may be directed, undirected, or a mixture of the two. Random variables are discrete, continuous, or mixtures. The network may be temporally static or dynamic.

Computations with PGMs are generally fall under the heading of inference or learning. The inference problem is well studied, and there exist many algorithms for exact and approximate inference with varying assumptions about the underlying network. The learning problem uses data (instances of random variables) to infer properties of the model. Interesting properties include probability distributions (parameter learning) and network structure (structure learning).

A PGM library has the (rather broad) goals of:

- ability to represent a wide variety of models
- easily add new algorithms

- useful to both researchers and practitioners

This works presents a rough design for such a library, unimaginatively called GM. The current implementation only provides a small subset of the desired functionality—inference in discrete, static graphical models. Rather than describing complete library implementation (or even a complete design), this work serves as a case study of language features useful for implementing a PGM library.

In particular, we focus on how features of ML serve to help or hinder constructing such a library. The implementation language is the ML variant Standard ML (SML) [8]. The SML module system is used extensively in the library.

Section 2 discusses the relevant language features of SML. Section 3 describes the GM library design, implementation, and issues regarding SML as the implementation language. Finally, Section 4 provides some concluding remarks.

2 Standard ML Requisites

SML is a modern functional language with an advanced type and module system. While the language encourages a functional style, SML includes support for impure features such as imperative assignments, exceptions, and continuations.

SML is a strongly typed language. Type checking in performed at compile time; type inference is used so that explicit type annotations are not required. Functions can be written that operate on arbitrary types—these functions are said to be polymorphic. The strong typing mechanism allows the language to guarantee that a compiled program will not crash due to runtime type errors.

Dynamic typing can be mimicked by explicitly constructing aggregate types which are the union of the desired types. Functions which operated on such type unions then have explicit cases for each type. This concept will be illustrated in the next section when describing the GM library.

2.1 The SML Module System

The SML module system provides a mechanism to divide a program into separate units with clearly defined interfaces. This section provides a condensed summary of relevant features of the SML module system. Harper [4] provides a gentle yet comprehensive coverage of SML in general and the module system in particular.

Signatures and structures are the two fundamental constructs in the module system. Signatures are used to describe interfaces. Structures provide the implementation. In this sense, signatures can be called the "type" of structures.

SML signature's provide an explicit definition of a structure's contents. These include declarations of sub-structures, exceptions, types, and values. Roughly

speaking, a structure matches a signature if it contains all of the elements (with the correct identifiers and types) of the signature.

The signature/structure deceleration can be used to express object oriented design patterns by declaring (in a signature) a single type (the object) and functions which operate on this type (member functions). Private member functions are not included in the signature. Object inheritance is implemented by means of signature inclusion. Child signatures are supersets of their parents—providing extended functionality. This technique is used in the GM library.

Functors are parameterized structures, which take other structures as arguments. Functor definitions are abstract—they must be invoked with the appropriate structure arguments to produce a tangible structure. The signatures of structures produced from a single functor can vary (with the parameters) between functor invocations.

Development with functors works with both top-down and bottom-up design approaches. In the bottom-up approach simple structures are first constructed. The smaller units are then combined via functor applications to create more complex structures.

A top-down approach begins by constructing functors to solve main goals. These functors are parameterized by structures which solve sub-goals. The code can be type-checked, but the functors cannot be applied to create a concrete implementation.

2.2 Development Environment

Several SML implementations are available—this library has been developed under the SML of New Jersey (SML/NJ) distribution. SML/NJ provides a reasonable efficient compiler which generates machine code. The GM code base conforms to the SML 97 specification, and so should compiler under other distributions such as MLton and Poly/ML.

A debugger for SML/NJ was developed in 1992 [2]. Unfortunately, recent releases of SML/NJ lack debugging facilities.

Debugging without a symbolic debugger can be managed with a combination of techniques. Firstly, the lack of a debugger encourages the use of concise functions that can be verified by inspection. The use of side-effect free functions helps with this reasoning process. SML/NJ (in fact, most distributions) provides an interactive environment which allows interaction via a read-eval-print loop. This is useful in the development/debugging process as the user can examine values and invoke functions.

3 Library Design

This section describes the design and (partial) implementation of the GM library. Emphasis is placed on the design with respect to the the SML module system. The majority of the code in GM is purely functional, however a few imperative features are used.

A major drawback of SML is lack of external librarys. This results in a diversion of time into developing support structures for the GM library. In particular, there was a need to develop routines for graph manipulation and multi-dimensional arrays. Both the graph and multi-dimensional array implementations are purely functional.

3.1 Modules

GM uses the SML module system to divide the library into independent units. Each module includes all of the modules on which it depends as sub-structures. This allows each module to be independent of the rest of the library (given its contents). The sub-structures of a module provide an explicit definition of the module's dependencies, and allow each module to be self contained.

The major modules of the GM library are:

- RVar: for representing random variables
- Potential: for specifying potentials
- CPD: for specification of CPDs
- LocalFactor: a local factor, the union of Potential and CPD
- IPotential: internal representation of potentials
- GModel: representation of a graphical model
- Inference: engines for performing inference

These modules are implemented as functors which take requisite substructures as parameters. The GM library provides a structure (called `GM`) which collects all of the functor applications into a single structure.

More information about the interfaces can be found in the signature files (.sig) contained in the GM distribution [9]

3.2 Sharing Considerations

The use of sub-structures in signatures creates a problem for the ML type system. Structures which are include included in multiple locations are treated as disparate, and so their types are not interoperable. This is problem solved by annotating signatures with a structure sharing declaration which makes desired interoperability explicit.

The sharing declaration is best explained by example. The following snippet exposes a part of the `GMODEL` and `LOCAL_FACTOR` signatures:

```
signature GMODEL =
sig
    structure LocalFactor : LOCAL_FACTOR
    structure RVar : R_VAR
    ...
end
signature LOCAL_FACTOR =
sig
    structure RVar : R_VAR
    ...
end
```

A graphical model needs to include sub-structures for local factors and random variables. A local factor will also need a substructure for random variables. However, this declaration provides no guarantee to the SML type system about the interoperability of these two RVar structures. Namely the GModel.RVar structure cannot operate on the values produced from the LocalFactor.RVar structure and vice versa. Any code which tries to do so will cause a compile-time type error.

This problem is solved by introducing sharing decelerations. The SML keyword `sharing` follows a group of structure definitions to denote which structures (ascribing a common signature) are actually the same. The GMODEL signature would be declared as follows:

```
signature GMODEL =
sig
    structure LocalFactor : LOCAL_FACTOR
    structure RVar : R_VAR
    sharing LocalFactor.RVar = RVar
    ...
end
```

The GM signature (which provides the top-level interface to the library) contains sharing specifications to assure that all of the structures which it provides can be used together (e.g they all use the same RVar structure).

3.3 An Example

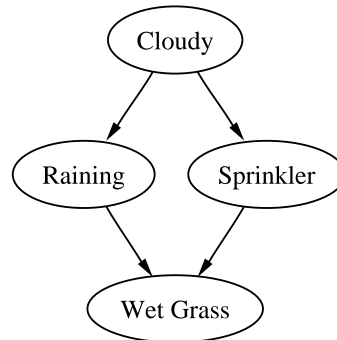

Figure 1: A simple example: Clouds exert causal influence on sprinkler and rain, which in turn influence wet grass. This figure was produced with `GModel`'s `toDotString` function.

This section presents a simple example of inference in the graphical model of Figure 1. First, the random variables are created, and given names:

```
val [FALSE, TRUE] = List.map RVar.intToValue [0,1]
val [C,S,R,W] = List.map RVar.newVar [2,2,2,2]
val _ = RVar.giveName C "Cloudy"
val _ = RVar.giveName S "Sprinkler"
val _ = RVar.giveName R "Raining"
val _ = RVar.giveName W "Wet Grass"
```

Variable naming (`RVar.giveName`) uses imperative assignment to update a table of identifier/name pairs. This list is used by the RVar structure when printing names.

Next, the local factors are created as conditional probability distributions:

```
val factors = List.map (LocalFactor.CPD o CPD.make)
              [([0.5, 0.5], C, []),
               ([0.8, 0.2, 0.2, 0.8], R, [C]),
               ([0.5, 0.5, 0.9, 0.1], S, [C]),
```

```
                ([1.0, 0.0, 0.1, 0.9,
                  0.1, 0.9, 0.01, 0.99], W, [S,R])]
```

Local factors can be either conditional probability distributions or potentials, so the LocalFactor type is a union of the two constituent types (CPD and Potential).

Finally, we construct the graphical model from a list of local factors, enter some evidence, and perform inference:

```
val gmodel = GModel.make local_factors
val _ = Util.showDot (GModel.toDotString gmodel)
val engine = Eng.makeEngine (gmodel, [],[] )
val engine = Eng.enterEvidence engine (Eng.Hard (W, TRUE))
val P_S = Eng.query engine [S]
val P_SR = Eng.query engine [R, S]
```

GM displays graphs using the dot language [6]. Structures define a `toDotString` method (where appropriate) which can be displayed with the `Util.showDot` method.

More examples are available with the GM distribution [9].

3.4 Testing

SML/NJ provides the CM structure [3] which provides a convenient mechanism for managing the compilation process of SML projects. This is used in GM to control both building and unit testing. Tests are written as structures which, through the compilation process, instantiate, run, and verify parts of the library.

Integrating testing into the build system allows for incremental testing approach. After a modification of the code base, the command

```
- CM.make "tests.cm";
```

is issued. This causes the appropriate modules to be recompiled. Any of the testing modules which depend on changed library models will also be recompiled. The process of recompiling the test modules runs the testing scripts, and outputs the results to the command line. This provides an immediate means of checking the correctness of changes made. Of course, this requires that appropriate testing modules are written. Currently GM has modules to test the multidimensional tables and inference engines.

3.5 Efficiency

A preliminary experiment was done to compare the speed of GM with BNT [10] (written in MATLAB). The model tested was a the ALARM network [1] which

consists of 37 discrete random variables ranging in dimension from 2 to 4. Join tree build time was measured (as the mean of 100 runs) for both cases libraries. BNT does not support queries which contain variables from multiple cliques, so querying was tested on a single variable (again as the mean of 100 runs). GM implements HUGIN style message passing [5].

Library	Build time (sec)	Query time (sec)
GM	0.2816	0.0001
BNT	0.7637	0.0019

Table 1: Runtime comparison of GM and BNT for building and querying a join tree.

These results confirm that SML can offer a speedup over MATLAB. This is hardly surprising as SML/NJ is compiled to native code while MATLAB is interpreted.

4 Conclusion

A library design for probabilistic graphical models was presented. A partial implementation in SML of this design was described. There are a number of properties of SML which are useful for implementing a graphical models library. Benefits of using SML include: strong typing and type inference, high level programming and an interactive development environment.

Strong typing combined with type inference allows the compiler to immediately detect nonsensical pieces of code. This allows many bugs to be caught at compile time. It is estimated that static typing alone caught 80% of the bugs in GM at compile time.

Coding with high-level constructs encourages the concise definition of many of the methods implemented in GM. A particularly useful technique was to define algorithms in terms of folds over data structures. An interactive environment facilitates the development in this style.

Several drawbacks of using SML were identified. The most important limitation of SML from the viewpoint of developing a graphical models library is the lack of library support for numerical, and particularly statistical, computations. Extending GM to support continuous variables would require implementation of additional structures to provide basic statistical functionality. In this regard, the ML dialect OCAML is more favorable as it is often used for numerical computations.

Another major drawback encountered in SML/NJ is the lack of a symbolic debugger. Other SML implementations (i.e. PolyML) provide debugging facilities, but GM uses features (i.e. the CM structure) which are only available in SML/NJ.

It may have been worthwhile to develop this library in a monadic style. Monads are a construct which allow purely functional languages to handle state [11]. In addition to handling imperative features such as in-place array updating in a functionally-pure manner, monads could be used to obtain many of the benefits of Aspect Oriented Programming (e.g. a general tracing facility for debugging) [7].

References

[1] I. Beinlich, H. Suermondt, R. Chaves, and G Cooper. The ALARM monitoring system: A case study with two probabilistic inference techinques for belif networks. In *Second European Conference on Artificial Intelligence in Medicine*, 1989.

[2] Att Bell. Debugging in Standard ML of New Jersey, 1992.

[3] Matthias Blume. *CM: The SML/NJ Compilation and Library Manager*, May 21 2002.

[4] Robert Harper. Programming in Standard ML. Available at : `http://www-2.cs.cmu.edu/~rwh/smlbook/online.pdf`, 2004.

[5] F. V. Jensen, S. L. Lauritzen, and K. G. Olesen. Bayesian updating in causal probabilistic networks by local computations. In *Computational Statistics Quaterly*, 1990.

[6] Eleftherios Koutsofios and Stephen C. North. *Drawing graphs with dot*. Murray Hill, NJ, 1996.

[7] Wolfgang De Meuter. Monads as a theoretical foundation for AOP. In *International Workshop on Aspect-Oriented Programming at ECOOP*, 1997.

[8] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. The definition of Standard ML (revised). 1997.

[9] Clint Morgan. Gm : source and documentation. Available at: `www.cs.ubc.ca/~clint/gm-sml`, 2004.

[10] Kevin P. Murphy. The Bayes Net Toolbox for MATLAB, 2001.

[11] P. L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78, New York, NY, 1990. ACM.